(1)

AFIT/GE/ENG/92D-30

DIGITAL SIGNAL PROCESSING USING LAPPED TRANSFORMS WITH VARIABLE PARAMETER WINDOWS AND ORTHONORMAL BASES

THESIS

Brian Dean Raduenz
Captain, USAF

AFIT/GE/ENG/92D-30

AFIT/GE/ENG/92D-30

DIGITAL SIGNAL PROCESSING USING LAPPED TRANSFORMS WITH VARIABLE PARAMETER WINDOWS AND ORTHONORMAL BASES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Brian Dean Raduenz, B.S.
Captain, USAF

December, 1992

Approved for public release; distribution unlimited

Preface

An undertaking of this magnitude is not successfully completed without the guidance and support of many others.

First and foremost, I would like to extend a special thanks to my thesis advisor, Dr. Bruce Suter. His own enthusiasm for the project was truly inspiring, and his knowledge and insight was crucial to my efforts.

Thanks also to my committee members, Maj Mark Mehalic and Dr Mark Oxley, for their constructive comments and corrections; Maj Eric Christensen, for providing crucial support of my Ada coding efforts; Dr Matthew Kabrisky, for giving freely of his time and broad engineering knowledge; Dr Tim Anderson, John Colombi, and Janet Slifka for their help in the Biocommunications Lab; and Pam Young for brightening many otherwise dismal days.

Finally, I must thank my wonderful parents, whose love and devotion made me the great guy that I am today.

Brian Dean Raduenz

Table of Contents

Page

List of Figures

AFIT/GE/ENG/92D-30

Abstract

This thesis develops and evaluates a number of new concepts and tools for the analysis of signals using variable overlapped windows and orthonormal bases. Windowing, often employed as a spectral estimation technique, can result in irreparable distortions in the transformed signal. By placing conditions on the window and incorporating it into the orthonormal representation, any signal distortion resulting from the transformation can be eliminated or cancelled in reconstruction. This concept is critical to the theory underpinning this thesis.

As part of this evaluation, a tensor product based general N-point fast Fourier transform algorithm was implemented in the DOD standard language, Ada. The most prevalent criticism of Ada is slow execution time. This code is shown to be comparable in execution time performance to the corresponding FORTRAN code. Also, as part of this thesis, a new paradigm is presented for solving the finite length data problem associated with filter banks and lapped transforms. This result could have significant importance in many Air Force applications, such as processing images in which the objects of interest are near the borders. In addition, a limited number of experiments were performed with the coding of speech. The results indicate the lapped transform evaluated in this thesis has potential as a low bit rate speech coder.

DIGITAL SIGNAL PROCESSING USING LAPPED TRANSFORMS WITH VARIABLE PARAMETER WINDOWS AND ORTHONORMAL BASES

I. Introduction

Digital signal processing is a vast engineering discipline primarily concerned with the representation of signals as a sequence of numbers or symbols, and the subsequent processing of these numbers toward some goal (25). One purpose for this processing is to transform a signal into a form which is in some sense more effectively transmitted as part of a communications system. *Decomposition* is one method of representing a signal in this more desirable form, while the recovery of the signal from this form is called *reconstruction.* Figure 1.1 depicts such a system. The actual form of the decomposition and reconstruction algorithms used in this thesis will be outlined in the next chapter.

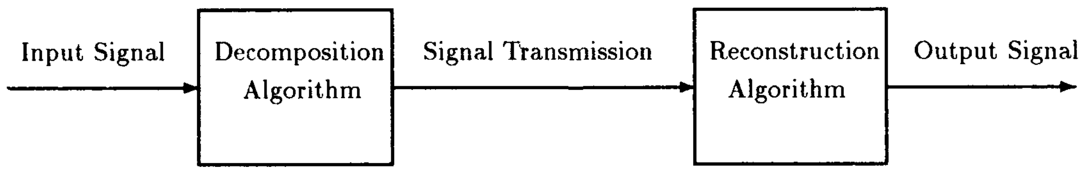

Figure 1.1. Communications System

1.1 Windows in Digital Signal Processing

The Fourier transform is indeed the cornerstone of modern signal processing and involves representation of a signal using an orthogonal sinusoidal basis set. In practice, every signal to be processed must be of finite length, and this length defines the minimum frequency spacing required for adjacent basis elements of a Fourier representation. When the data is transformed, any frequencies not commensurate with one of the basis frequencies will exhibit non-zero projections on all basis frequencies (14:52). This phenomenon is known as spectral leakage. If the primary goal of a system is harmonic analysis, spectral leakage is especially incommodious and can be lessened through the use of *windows* specifically designed for this purpose. Harris provides a thorough discussion on this subject in (14). When windows are used as a step in decomposing a signal for eventual recovery, however, the properties and parameters of the windows to be used need to be quite different from those designed specifically to limit spectral leakage. While spectral leakage may be apparent in the decomposed signal, the primary concern of the decomposition/reconstruction system is proper reconstruction of the original signal. Toward this end, Princen and Bradley (27) introduced a technique whereby the window is an integral part of the orthonormal basis. This less traditional use of windows will be shown to be an important concept toward the development of this thesis.

1.2 Perfect Reconstruction

Any algorithm which is developed as part of a decomposition/reconstruction scheme attempts to reconstruct a signal which is as close as possible to the original input. If all decomposition coefficients are used in the reconstruction, and the reconstructed signal is an exact copy (with the possibility of a delay) of the original signal, then the system satisfies the perfect reconstruction property (35). Satisfying this property is an important requirement for any decomposition/reconstruction scheme (27:1153), so that coding distortion resulting from compression techniques have a minimal impact on reconstruction of the original signal.

1.3 Research Goals

Several goals were established which guided the research of this thesi. All of the goals in some way involve the Generalized Lapped Orthonormal Transform (GLOT), which will be outlined in detail in Chapter 2. These goals are outlined below:

1. Implement the GLOT in software and validate the theory outlined in Chapter 2, including perfect reconstruction using variable size windows and overlap.

2. Investigate use of the GLOT in processing finite length signals without the boundary effects associated with traditional approaches.
3. Investigate use of the GLOT in speech processing. Specifically, determine the compression capability of the GLOT using various spectral compression techniques.

The first goal is necessary because this is the first research regarding the GLOT. After implementing the complete software program, perfect reconstruction will be validated for different size windows and overlap. Processing finite length signals often results in distortions at the signal boundaries. The second goal is to propose a method of achieving perfect reconstruction for this type of signal. Finally, the third goal is a logical extension of previous work invloving speech and lapped transforms, most notably by Malvar (18).

1.4 Speech Compression - An Application

One area which will be investigated is compression of speech data, since something other than a simple analog-to-digital conversion is required to make transmitting speech economical in terms of bit rate. Compressing speech requires a three way trade-off among the goals of preserving intelligibility and quality, limiting the bit rate, and minimizing computational complexity (26:225). The impact of any compression technique on these three often incompatible goals must be continually assessed in determining the most promising compression algorithm. Transform coding is one of the major low bit rate speech coding techniques being investigated by the military, since channel bandwidth is extremely limited in many strategic and tactical systems (38). This thesis will include preliminary experiments using the GLOT that will be useful in describing the design space for a low bit rate speech encoder.

1.5 Outline

This chapter provides an introduction to this thesis, describing the goals of this thesis and particular field of research within the vast discipline of digital signal processing. The background and theory which form the basis for this thesis, up to and including the GLOT, is outlined in Chapter 2. Chapter 3 provides a complete derivation and analysis of an Ada based fast Fourier transform algorithm, the most significant coding effort of this thesis. Chapter 4 provides the link between the theory of Chapter 2 and the application of this theory to various signal processing and communications problems, including finite length processing, and speech processing and compression. Finally, a summary of key results, and a recommendation for follow-on research areas, are outlined in Chapter 5.

II. Background

In (33), Suter and Oxley provide the derivation of a new method for the decomposition and reconstruction of signals based on variable parameter windows and weighted orthonormal basis functions. The aspect of this derivation which is different from the conventional methods of transform coding is that the orthonormal basis used to represent the signal of interest is actually the product of a window and some type of orthonormal function (in this application, a sinusoidal function). As described in Section 1.1, this technique was developed to overcome some of the limitations of finite length processing. Malvar (18) (21) has investigated such a method, called the Lapped Orthogonal Transform (LOT), and adapted this approach to speech signals to overcome what he has termed "blocking effects." Combining the window and function to form an orthonormal basis leads to an overlap of adjacent intervals, and Malvar limited his analysis to constant length windows with a $50/k\%$ overlap, for positive integer k (19). He was able to eliminate the "extraneous tones" in speech resulting from the conventional windowing scheme (21:553). Coifman and Meyer (8) generalized Malvar's analysis to include variable size windows. Suter and Oxley's approach is based in part on these developments, and further generalizes previous work to include variable bases (including non-sinusoidal bases), variable length windows, variable overlap between adjacent windows, and variable window amplitude. They also provide a computationally efficient algorithm (33). The transformation outlined by Suter and Oxley will be mathematically defined in the next section.

Although not an integral part of this thesis, it is important to note that there are intimate relationships between multirate filter banks, lapped orthogonal transforms, and wavelets (2) (9) (20) (35).

The initial implementation, which forms the basis for this thesis, is restricted to a sinusoidal basis with a constant amplitude window of variable size and a variable percent overlap with adjacent windows. These initial assumptions will be used to tailor the derivation of the transformation outlined in (33) to the work encompassed herein.

2.1 The Generalized Lapped Orthonormal Transform

2.1.1 Mathematical Definition The following is a formal definition of the lapped transform used in this thesis.

Let **N** represent the natural numbers and **Z** represent the integers.

Let $\{a_j | j \in \mathbf{Z}\}$ be a sequence such that $a_j \to \infty$ as $j \to \infty$ and $a_j \to -\infty$ as $j \to -\infty$.

Let $\{\epsilon_j | j \in \mathbf{Z}\}$ be a sequence such that $\epsilon_j > 0$ and $\epsilon_j + \epsilon_{j+1} \leq a_{j+1} - a_j$.

Let $\{f_{j,k} | k \in \mathbf{N}\}$ be an orthonormal set on $[a_j, a_{j+1}]$ with respect to weight $p_j(x)$.

Let $w_j(x)$ be a window.

Then, define the Generalized Lapped Orthonormal Transform (GLOT) $\mathbf{T} : \mathbf{L}^2(\Re) \to \ell^2(\mathbf{Z} \times \mathbf{N})$.

$$[\mathbf{Ts}]_{j,k} = \int_{a_j - \epsilon_j}^{a_{j+1}+\epsilon_{j+1}} s(x) u_{j,k}(x) dx, \tag{2.1}$$

where $u_{j,k}(x) = w_j(x) \tilde{f}_{j,k}(x) \sqrt{\tilde{p}_j(x)}$.

The form of $w_j(x)$, $\tilde{f}_{j,k}(x)$, and $\tilde{p}_j(x)$ used in this thesis will be given when describing the signal decomposition in the following sections.

2.1.2 Signal Decomposition The decomposition portion of the GLOT involves, at the highest level, three main steps in the order outlined below:

1. Multiplication of the Signal by a Window
2. Folding of the Data
3. Performing a Fourier Transform

Each of these steps will be outlined followed by a summary of the full forward decomposition. First, however, some notation must be provided. Each interval or block of data to be processed will span the closed interval $[a_j, a_{j+1}]$ as shown in Figure 2.1. An orthonormal basis, $\{f_{j,k}(x) | k \in \mathbf{N}\}$, will be defined on this interval. This orthonormal basis must satisfy

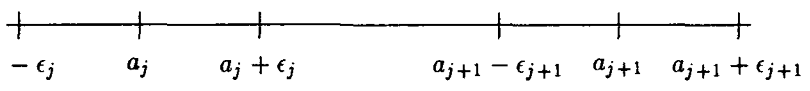

Figure 2.1. Partitioned Axis

the orthonormality property (33), given in Equation 2.2:

$$\int_{a_j}^{a_{j+1}} f_{j,k}(x) f_{j,l}(x) p_j(x) dx = \delta_{k,l}. \tag{2.2}$$

where j indexes the current interval, $f_{j,k}(x)$ specifies the kth basis function within the jth interval, and $p_j(x)$ specifies a weight function over interval j. The weight function (and its extension, $\tilde{p}_j(x)$) for a sinusoidal basis function is unity for all x. Because only a sinusoidal basis was implemented in this thesis, the weight function may be omitted from further equations.

The orthonormal basis used in this research was the generalized sinusoidal basis of Coifman and Meyer (8) defined by $f_{j,k}(x)$ on the interval $[a_j, a_{j+1}]$:

$$f_{j,k}(x) = \sqrt{\frac{2}{a_{j+1} - a_j}} \sin\left(\pi(k + 1/2)\left[\frac{x - a_j}{a_{j+1} - a_j}\right]\right). \tag{2.3}$$

The first three sinusoidal basis functions for $k = 0, 1, 2$ over a normalized interval are depicted in Figure 2.2. Next, the odd and even extensions of $f_{j,k}(x)$ are defined on $(a_j - \epsilon_j, a_j)$ and $(a_{j+1}, a_{j+1} + \epsilon_{j+1})$, respectively, resulting in a new function $\tilde{f}_{j,k}(x)$, defined by:

$$\tilde{f}_{j,k}(x) = \begin{cases} 0 & , \ -\infty < x \leq a_j - \epsilon_j \\ \sqrt{\frac{2}{a_{j+1} - a_j}} \sin\left(\pi(k + 1/2)\left[\frac{x - a_j}{a_{j+1} - a_j}\right]\right) & , \ a_j - \epsilon_j < x < a_{j+1} + \epsilon_{j+1} \\ 0 & , \ a_{j+1} + \epsilon_{j+1} \leq x < \infty. \end{cases} \tag{2.4}$$

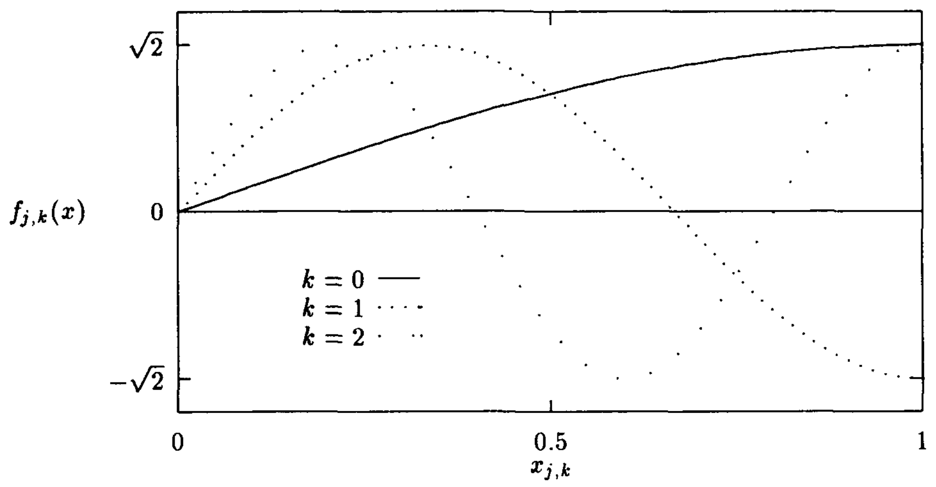

Figure 2.2. Sinusoidal Basis Functions: $k = 0, 1, 2$ over Normalized Interval

To guarantee that the interval $(a_j, a_j + \epsilon_j)$ does not overlap with $(a_{j+1} - \epsilon_{j+1}, a_{j+1})$, it is required that $\epsilon_j + \epsilon_{j+1} \leq a_{j+1} - a_j$. That is, if $\epsilon_j = \epsilon_{j+1}$, then the maximum value of ϵ_j is 50% of the length of the window. A general proof that the product of the window *and* function (where both satisfy given criteria) form an orthonormal basis may be found in (33).

The goal of this algorithm is to derive a set of coefficients which represent the data (coefficients which can then be used to reconstruct the original signal). In other words, the signal of interest will be expanded in terms of an orthonormal basis, $\{u_{j,k} | j \in \mathbf{Z}, k \in \mathbf{N}\}$. The orthonormal basis has been derived as the product of a window, w_j, and an expanded orthonormal basis function, $\tilde{f}_{j,k}(x)$. The signal expansion in terms of the resulting orthonormal basis (33) is given by:

$$s(x) = \sum_{j \in \mathbf{Z}} \sum_{k \in \mathbf{N}} \alpha_{j,k} u_{j,k}(x). \tag{2.5}$$

The forward transformation yields coefficients $\alpha_{j,k}$, which are defined specifically by:

$$\alpha_{j,k} = \int_{a_j-\epsilon_j}^{a_{j+1}+\epsilon_{j+1}} s(x) w_j(x) \tilde{f}_{j,k}(x) dx. \tag{2.6}$$

where $s(x)$ is the data to be represented, and $\tilde{f}_{j,k}(x)$ is as defined previously in Equation 2.4. The window function, $w_j(x)$, will be defined in the following section.

2.1.2.1 Multiplication of the Signal by a Window To uniquely specify the GLOT, a window must be defined. The window must have certain properties to ensure that the product of this window and the expanded orthonormal basis function also form an orthonormal basis set. Properties first defined by Coifman and Meyer (8) include:

$$\begin{array}{llcll} (a) & w_j(x) & = & 1 & \text{for} \quad x \in (a_j + \epsilon_j, a_{j+1} - \epsilon_{j+1}) \\ (b) & w_j(x) & = & 0 & \text{for} \quad x \notin (a_j - \epsilon_j, a_{j+1} + \epsilon_{j+1}) \\ (c) & w_j(a_j - \sigma) & = & w_{j-1}(a_j + \sigma) & \text{for} \quad \sigma \in [-\epsilon_j, \epsilon_j] \\ (d) & w_j^2(x) + w_{j-1}^2(x) & = & 1 & \text{for} \quad x \in [a_j - \epsilon_j, a_j + \epsilon_j] \end{array} . \tag{2.7}$$

One window satisfying these properties (see Suter and Oxley (33)) is given by:

$$w_j(x) = \begin{cases} 0 & , \ -\infty < x < a_j - \epsilon_j \\ \sin\left(\frac{\pi}{4\epsilon_j}\{x - [a_j - \epsilon_j]\}\right) & , \ a_j - \epsilon_j \le x \le a_j + \epsilon_j \\ 1 & , \ a_j + \epsilon_j < x < a_{j+1} - \epsilon_{j+1} \\ \cos\left(\frac{\pi}{4\epsilon_{j+1}}\{x - [a_{j+1} - \epsilon_{j+1}]\}\right) & , \ a_{j+1} - \epsilon_{j+1} \le x \le a_{j,+1} + \epsilon_{j+1} \\ 0 & , \ a_{j+1} + \epsilon_{j+1} < x < \infty. \end{cases} \tag{2.8}$$

This sinusoidal window was used for all research in this thesis and its asymmetrical overlap with the two adjacent windows is depicted in Figure 2.3. For these windows, the polynomial within the braces is an affine function of x. Coifman (7) has developed other windows where the polynomial within the braces is a higher degree polynomial function of x. Another window satisfying all properties in Equation 2.7 which has the additional advantage of being differentiable at $x = a_j - \epsilon_j$ and $x = a_{j+1} + \epsilon_{j+1}$ is defined by Suter and Oxley in (33).

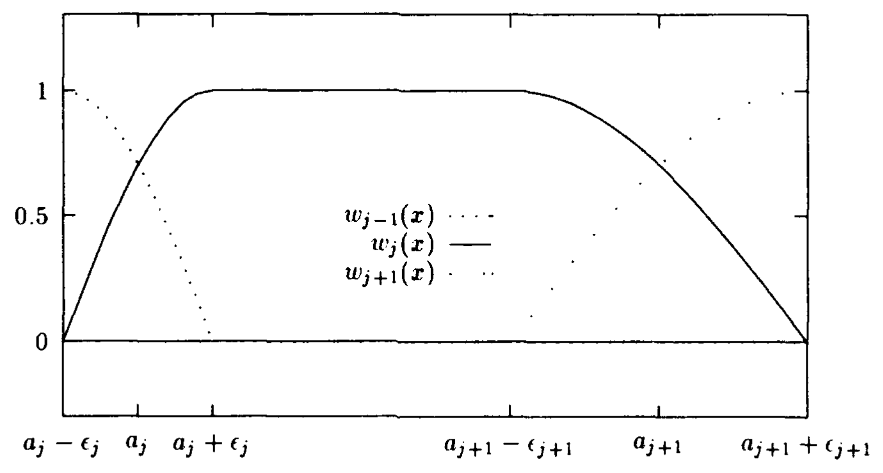

Figure 2.3. Example Window

2.1.2.2 Folding of the Windowed Data An important step in GLOT (33) is the folding (and unfolding discussed as part of the reconstruction) of the data to form a new vector, h_j, as shown in Equation 2.9 below:

$$h_j(x) = \begin{cases} s(x)w_j(x) - s(2a_j - x)w_j(2a_j - x) & , \quad a_j \leq x \leq a_j + \epsilon_j \\ s(x) & , \quad a_j + \epsilon_j < x < a_{j+1} - \epsilon_{j+1} \\ s(x)w_j(x) + s(2a_{j+1} - x)w_j(2a_{j+1} - x) & , \quad a_{j+1} - \epsilon_{j+1} \leq x \leq a_{j+1}. \end{cases} \tag{2.9}$$

The folding is a natural consequence of the mathematical proof for the orthonormal basis (33), and has several important implications. Consider the interval $(a_j - \epsilon_j, a_j + \epsilon_j)$, shown in Figure 2.4. To demonstrate the folding, the value $h_j(a_j - l)$ will be a sum of the original data at $a_j - l$ and $a_j + l$ multiplied by the appropriate window value (in this case, the window value indicated by arrows in Figure 2.4). Data in the interval $(a_j - \epsilon_j, a_j + \epsilon_j)$ will be included in the processing of data in intervals j *and* $(j - 1)$. Recall that the windowed overlapped orthonormal basis is designed to eliminate edge effects. The theory predicts it will be possible to reconstruct the data points in this region as a weighted sum or difference of two coefficients from adjacent intervals. The reconstruction algorithm will

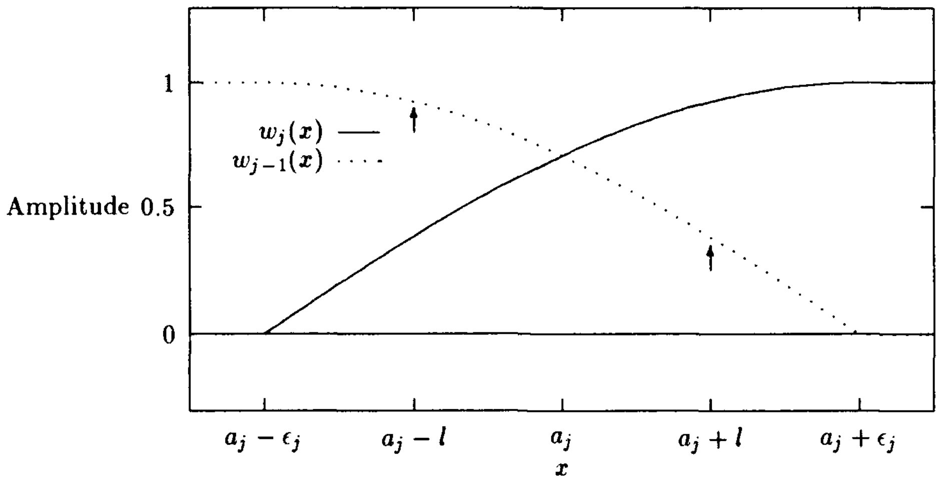

Figure 2.4. Folding of Data

be outlined in more detail in the Section 2.1.3. Henceforth, all subsequent processing of the data in the jth interval can be confined to $[a_j, a_{j+1}]$, even though this processing is affected by a larger number of data points, $[a_j - \epsilon_j, a_{j+1} + \epsilon_{j+1}]$

2.1.2.3 Fast Fourier Transform A fast Fourier transform (FFT) is employed in both the decomposition and reconstruction stages of the GLOT. The form of the FFT used is given by:

$$G_{j,k} = \frac{1}{M_j} \sum_{l=0}^{M_j-1} g_{j,l} e^{i2\pi kl/M_j}, \tag{2.10}$$

where vector $G_{j,k}$ is the one-dimensional transform of vector $g_{j,l}$, and M_j is the length of these vectors. Normally, this form is considered an inverse FFT based on the positive argument of the exponential kernel. The inverse FFT used in both the decomposition and reconstruction is of this same form, and it will subsequently be refered to simply as an FFT. Normally, the resulting complex vector from the FFT has significant real and imaginary parts. In the decomposition, only the real portion of the FFT output vector contains any information. In the reconstruction, the complex FFT output is scaled and only the imaginary portion of the scaled output vector contains any information.

2.1.2.4 Decomposition Algorithm Summary With the major steps in the decomposition algorithm outlined, a summary of the full forward transformation can be provided. The theory (33) allows arbitrary spacing between samples. This spacing has been assumed to be unity so that $x_{j,l} = a_j + l$ for $l = 0, 1, ..., N_j$ in the following algorithm summary:

(1) Obtain data $s(x_{j,l})$ for $a_j - \epsilon_j \leq x_{j,l} \leq a_{j+1} + \epsilon_{j+1}$,
that is, $l = -M_j, .., -1, 0, 1, ..., N_j + M_{j+1}$.

(2) Multiply $s(x_{j,l})$ by window $w_j(x_{j,l})$ for $l = -M_j, ..., -1, 0, 1, ..., N_j + M_{j+1}$.

(3) Fold the sequence by defining

$$H_{j,l} = \begin{cases} s(x_{j,l})w_j(x_{j,l}) - s(2a_j - x_{j,l})w_j(2a_j - x_{j,l}) & , \quad a_j \leq x_{j,l} \leq a_j + \epsilon_j \\ s(x_{j,l}) & , \quad a_j + \epsilon_j < x_{j,l} < a_{j+1} - \epsilon_{j+1} \\ s(x_{j,l})w_j(x_{j,l}) + s(2a_{j+1} - x_{j,l})w_j(2a_{j+1} - x_{j,l}) & , \quad a_{j+1} - \epsilon_{j+1} \leq x_{j,l} \leq a_{j+1} \end{cases}$$

for $l = 0, 1, ..., N_j$.

(4) Define new array

$$\beta_{j,l} = H_{j,l} \sin\left(\frac{\pi l}{2N_j}\right) \quad \text{for} \quad l = 0, 1, \ldots, N_j \quad .$$

(5) Define the even extension of $\beta_{j,l}$

$$\tilde{\beta}_{j,l} = \begin{cases} \beta_{j,l} & , \quad 0 \leq l \leq N_j - 1 \\ \beta_{j,2N_j - l} & , \quad N_j \leq l \leq 2N_j - 1. \end{cases}$$

(6) Perform an FFT of length $2N_j$ on $\tilde{\beta}_{j,l}$ and define $D_{j,k}$ by:

$$D_{j,k} = \mathcal{R}e\left(\frac{1}{2N_j} \sum_{l=0}^{2N_j - 1} \tilde{\beta}_{j,l} e^{i2\pi kl/2N_j}\right) .$$

(7) Interpret the results of the FFT

$$\begin{aligned}\alpha_{j,0} &= \sqrt{2N_j}D_{j,0} \\ \alpha_{j,k} &= \alpha_{j,k-1} + 2\sqrt{2N_j}D_{j,k} \quad .\end{aligned}$$

The coefficients after decomposition, $\alpha_{j,k}$, will be referred to as spectral coefficients or coefficients throughout this thesis. In the case of the speech application, any further processing to achieve compression of the signal will be performed at this stage in the spectral domain.

2.1.3 Signal Reconstruction The reconstruction algorithm is the mathematical inverse of the decomposition, although the reconstruction implementation does not mirror a reverse ordered decomposition. The reconstruction (33) involves three steps as detailed below:

(1) Define odd extension of $\alpha_{j,k}$ by:

$$\tilde{\alpha}_{j,k} = \begin{cases} \alpha_{j,k} & , \quad 0 \le k \le N_j - 1 \\ -\alpha_{j,2N_j-k-1} & , \quad N_j \le k \le 2N_j - 1 \end{cases}$$

(2) Perform an FFT of length $2N_j$ on $\tilde{\alpha}_{j,k}$ for $k = 0, 1, 2, \ldots, 2N_j - 1$ and define $H_{j,l}$ by:

$$H_{j,l} = \sqrt{2N_j}\mathcal{I}m\left(e^{i\frac{\pi l}{2N_j}}\left[\frac{1}{2N_j}\sum_{k=0}^{2N_j-1}\tilde{\alpha}_{j,k}e^{i\frac{2\pi kl}{2N_j}}\right]\right)$$

(3) Reconstruct the signal on the interval $[a_j, a_{j+1}]$ at the data points $x_{j,l} = a_j + l$ for $l = 0, 1, 2, \ldots, N_j$ by using

$$S_{j,l} = \begin{cases} \dfrac{H_{j-1,l}}{2w_{j-1}(x_{j,l})} & , \quad x_{j,l} = a_j \\ w_j(2a_j - x_{j,l})H_{j-1,N_{j-1}-l} + w_j(x_{j,l})H_{j,l} & , \quad a_j < x_{j,l} < a_j + \epsilon_j \\ H_{j,l} & , \quad a_j + \epsilon_j \le x_{j,l} \le a_{j+1} - \epsilon_{j+1} \\ w_j(x_{j,l})H_{j,l} - w_j(2a_{j+1} - x_{j,l})H_{j+1,N_j-l} & , \quad a_{j+1} - \epsilon_{j+1} < x_{j,l} < a_{j+1} \end{cases}$$

The values $S_{j,l}$ will approximate the signal evaluated at $x_{j,l}$, that is, $S_{j,l} \approx s(x_{j,l})$.

There are a few noteworthy differences as compared to the decomposition algorithm. First, only the imaginary (rather than the real) portion of the complex scaled FFT result is of interest. Second, the unfolding and division by the window are combined in one step. Notice that each reconstructed point in a region of overlap requires the contribution of two different windows and coefficients. This is consistent with the fact that spectral coefficients were derived after folding windowed data points from adjacent windows (see Equation 2.9 above).

If all spectral coefficients from the forward transformation are used in the reconstruction, perfect reconstruction should result. In this case, $S_{j,l} = s(x_{j,l})$.

2.2 Finite Length Signal Considerations

A common signal processing problem involves proper reconstruction of finite length signals. When performing sub-band analysis using multirate filter banks, for example, the total amount of data required in the channel is greater than that of the finite length input (and reconstructed output) (16:162). Assumptions must be made about the data beyond the signal extent, which may create errors in the reconstructed output. The most common methods being investigated are various signal extensions (37), including zero padding, periodic extension, symmetric extension, and boundary value replication. In practice, all signals must be of finite extent. This problem is especially significant, however, in applications such as image coding (1) (5) (23), time-frequency analysis (17), and matrix computations (3) (4).

The GLOT is also susceptible to this problem, since overlapping is employed and data on $[a_j - \epsilon_j, a_{j+1} + \epsilon_{j+1}]$ is required to process and reconstruct data on $[a_j, a_{j+1}]$. There is currently no lapped-transform based solution to processing a finite length signal which achieves perfect reconstruction without storing additional data outside the interval of interest, although some approaches based on approximations have been presented (10). This section will outline a solution to the problem which employs the GLOT.

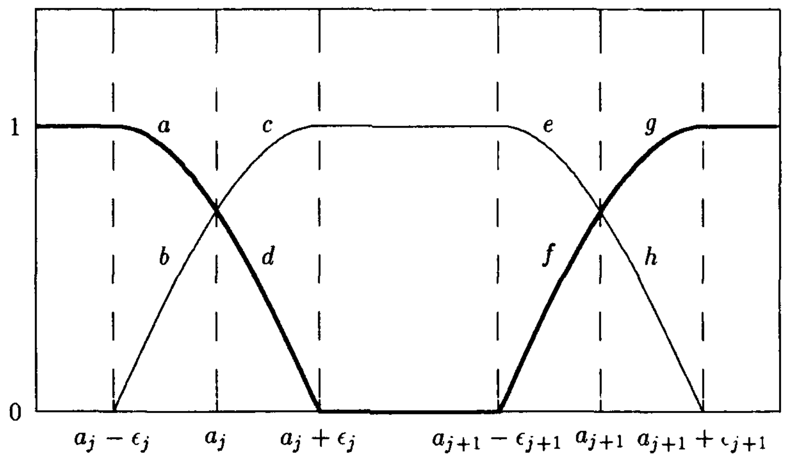

Figure 2.5. Periodically Extended Interval $[a_j, a_{j+1}]$

Before discussing a solution to the finite length signal problem, three goals will be outlined. First, perfect reconstruction of the entire signal is required. This goal assumes that the data at the boundaries must be preserved as well as the rest of the data. Second, the transform employed should be nonexpansive (5), meaning no extra data must be stored outside the interval $[a_j, a_{j+1}]$, thereby conserving data storage costs. Third, the complexity of the method should be no more than for the GLOT outlined in the previous sections. These three goals will be used to measure the success of the solution provided below. Although no additional data points will be required, it will be convenient to initially visualize the periodic extension of the window and data before and after the finite interval. Now, treat the boundary points of the interval $[a_j, a_{j+1}]$ as midpoints of the transition regions of the same percent overlap. As illustrated in Figure 2.5, there is a correspondence between the overlap at the two boundaries. In other words, the following window segments are identical and span an interval where the data is also identical: a and e, b and f, c and g, d and h. Because of this correspondence, only the window segments c, d, e, and f and data from a_j to a_{j+1} are required to reconstruct data on $[a_j, a_{j+1}]$. For example, the product of window segment a and the data on $[a_j - \epsilon_j, a_j]$ can be replaced by window segment ϵ and data on $[a_{j+1} - \epsilon_{j+1}, a_{j+1}]$ in the GLOT equations given above. This satisfies the second goal of a nonexpansive transform. It should be noted that perfect reconstruction can be achieved by storing data other than the periodic extension outside the interval of interest. One example would be to zero-pad outside $[a_j, a_{j+1}]$. Perfect reconstruction could be achieved, however, there would be an additional requirement to store the zeroes at specific locations. There is nothing inherent in this method which limits the reconstruction capabilities of the GLOT, including the potential discontinuity in the data at the boundary. Also, the complexity of this solution is no greater than that for the GLOT. All of the same transformation steps are followed, but with data on different intervals. This solution should satisfy all three goals outlined above. Not only is this idea a fairly simple theoretical concept, it will be shown in Chapter 4 to be a very practical approach to solving the problem.

2.3 Independent Interval Processing

The process outlined above solves the problem of processing the first and last segments of a finite length record of data. This concept could be expanded and applied to each interval of data, rather than just at the signal boundaries. If the window and data in the jth interval were periodically extended as shown in Figure 2.5, then processing the jth interval would be independent of the data in either adjacent window. Such a system would provide a decomposition/reconstruction algorithm achieving perfect reconstruction on an arbitrary length data, without requiring any overlap between adjacent intervals. This type of nonexpansive signal processing has not previously been defined for any lapped transform. Chapter 4 will provide the results of using the GLOT for this type of independent interval processing.

III. Development and Analysis of an Ada Based Fast Fourier Transform

The most obvious and important initial coding task was development of a fast Fourier transform (FFT) routine for a general number of input points, in accordance with step 6, Section 2.1.2.4, and step 2, Section 2.1.3. This routine was seen as the most formidable in terms of software implementation, and was consequently the highest coding priority. This chapter outlines the implementation of the FFT used in this thesis, and is based on the work by Raduenz, *et al.* in (29).

3.1 Choice of a Software Language

Because the choice of a language in thesis software development is significant, a few remarks on this choice are in order. All software developed during the course of this thesis was programmed in the high-level language Ada. Ada was designed and developed by the Department of Defense in the late 1970's and early 1980's to ensure sound software engineering concepts were employed in the development of military systems. The Ada advantages of modularity, readability, testability, and reliability were paramount to the success of the coding phase of this thesis. Further, follow-on students or any interested programmer may use the packages developed for this thesis, even by calling them from another language such as *C* or FORTRAN.

3.2 Background

Using tensor analysis, Ferguson (12) presented an elegant derivation of Glassman's (13) general N fast Fourier transform (FFT), together with a concise FORTRAN program to implement this FFT. Moreover, most FFT routines that were developed after Ferguson's work (12) were and continue to be based on tensor analysis (see, for example Van Loan (36)). Nonetheless, most FFT routines commercially available today operate on a vector of length $N = 2^m$, where m is an integer. The GLOT, however, requires a wider choice of N for general implementation. One of the practical advantages of Glassman's routine is that it can be used in digital signal processing applications for analysis of data of arbitrary length, without the coding complexity of Singleton's case driven routine (30). The principal disadvantage of Glassman's routine is that it requires an N-Vector working space. When programmed in Ada, however, the implementation details of this and other aspects of the code can be hidden in the body of an FFT package (making them transparent to the user), as shown in the next section.

Industry and laboratories have been somewhat lethargic about commercial use of Ada, although today the market approaches $1.5 billion annually (11). The military has also been criticized for allowing limitless waivers and generally impeding the transition to Ada. Many feel Ada's main disadvantage is slow execution time, thereby rendering it not applicable to many engineering applications. After delineating the conversion of Ferguson's FORTRAN program to Ada, an execution time comparison will demonstrate that Ada execution time is competitive with Ferguson's FORTRAN FFT execution time.

3.3 Source Code Comparison

Figure 3.1 is a listing of Ferguson's FFT subroutines. These subroutines were stored and compiled as separate files. Figure 3.2 gives the analogous Ada code in a FFT package specification and body. The source code in the two languages is dissimilar in a few notable areas. First, the FORTRAN program allows implicit assignment from a single index array to a three index array within the *Glassman* subroutine. This assignment is handled by the compiler. In Ada, the same assignment could be coded as a triple nested loop, assigning each element in *Data_Vector* to the appropriate location in a new three dimensional vector. A faster and more elegant assignment can be used, however, if the appropriate element in the single index vector can be accessed based on the values of the three indices. It can be shown that some variable *Data_Vector_3Index*(i, j, k), for $i \in [1, A], j \in [1, B], k \in [1, C]$ can be represented as *Data_Vector*$((i-1)*B*C+(j-1)*C+k)$ where A, B and C are the integers passed to *Glassman*. The above expression can be simplified to :

$$Data\_Vector(((i-1)*B+(j-1))*C+k). \qquad (3.1)$$

The assignments shown in Figure 3.2 were derived by substituting the appropriate indices into equation 3.1. When writing to the vector *Temp*, which becomes the output, the indices of the three index vector used in Figure 3.1 match the order of the nested loops, and this allows for an Ada assignment via a simple counter variable.

Second, the user of the Ada version need only pass a complex data vector and boolean inverse operator to *FFT*. Work space is established within the appropriate subprocedure, and the vector length N can be determined using Ada array attributes, thereby eliminating two of the passed parameters. Because work space in Ada is not defined at the top level, there is no need to call *Glassman* using a boolean operator and alternating if statements (passing either *U* or *Work* as the input vector), as is done in the FORTRAN routine. The only reason to pass two arrays would be to maintain integrity of the input data while passing data out as a separate vector.

Finally, the Ada code contains package calls to *Complex_Package, Type_Package,* and *Math* for standard complex number manipulations, global type declarations, and mathematical operations respectively, as well as a $1/N$ scaling at the end of subprocedure *FFT* for the inverse transform. All timing analysis was performed using the forward transform so no scaling would be invoked in either routine. In terms of floating point operations, the Ada and FORTRAN routines are equivalent.

3.4 Execution Time Comparison

Execution times for the two programs were compared using CPU time from the Unix *time* command. Although elapsed CPU times are measured to the 50th second with this facility, the exact execution time for the FFTs is not of great importance and is highly machine dependent. The desired result was a relative measure of FORTRAN and Ada execution time for digital signal processing algorithms such as the one used here.

Runs were made on a Sun SPARCstation 2 (operating system version SunOS 4.1.2), with very light additional load. The software packages used in this comparison were Sun FORTRAN (Enhanced FORTRAN 77) Version 3.1.1 and Verdix Ada Version 6.0.3(d). The results are given in Table 3.1. Notice that executables were developed which provided no output or wrote to a file to isolate the effects of I/O in the comparison. As is shown, the Ada execution times are comparable to the FORTRAN execution times.

```
      subroutine FFT_Sub (N, U, Work, Inverse)
          integer N
          complex U(N), Work(N)
          logical Inverse
          integer A, B, C
          logical Inu
          A    = 1
          B    = N
          C    = 1
          Inu  =. true.
C
10        if (B .GT. 1) goto 30
          if (Inu) return
          do 20 i = 1, N
              U(i) = Work (i)
20        continue
      return
30        A = C * A
          do 40 C = 2, B
              if (mod(B,C) .EQ. 0) go to 50
40        continue
50        B = B / C
          if (Inu)          call Glassman(A, B, C, U, Work, Inverse)
          if ( .Not. Inu)   call Glassman(A, B, C, Work, U, Inverse)
          Inu = .Not. Inu
          go to 10
      end
C-----------------------------------------------------------------
C-----------------------------------------------------------------
      subroutine Glassman (A, B, C, Uin, Uout, Inverse)
          integer A, B, C
          complex Uin(B,C,A), Uout(B,A,C)
          logical Inverse
C
C         This subroutine is called from FFT_Sub
C
          Complex Delta, Omega, Sum
          Twopi    =    6.28318530717958
          Angle    =    Twopi / float(A*C)
          Delta    =    CMPLX(Cos(Angle), -Sin(Angle))
          if (Inverse)  Delta = CONJG (Delta)
C
          Omega = CMPLX(1.0,0.0)
          do 40           IC = 1, C
              do 30       IA = 1, A
                  do 20   IB = 1, B
                      Sum = Uin(IB,C,IA)
                      do 10 JCR = 2, C
                          JC = C + 1 - JCR
                          Sum = Uin(IB,JC,IA) + Omega * Sum
10                    continue
                      Uout(IB,IA,IC) = Sum
20                continue
                  Omega = Delta * Omega
30            continue
40        continue
C
      return
      end
```

Figure 3.1. Fortran Code for FFT Subroutines

```
with Complex_Pkg;           use Complex_Pkg;
with Type_Package;          use Type_Package;
with Math;                  use Math;

package FFT_Pack is
    procedure FFT ( FFT_Data            : in out Complex_Vector ;
                    Inverse_Transform   : in boolean              );
end FFT_Pack;
------------------------------------------------------------------------------
------------------------------------------------------------------------------
package body FFT_Pack is
    procedure Glassman ( A, B, C              :    in integer;
                         Data_Vector          :    in out Complex_Vector ;
                         Inverse_Transform    :    in boolean     ) is

            Temp                 : Complex_Vector(1..A*B*C);
            Counter,JC           : integer := 0;
            Two_Pi               : constant float := 6.28318530717958;
            Del, Omega, Sum      : Complex;
            Angle                : float;
            C_Plus_1             : integer := C + 1;

        begin
            Angle    := Two_Pi / (float(A*C));
            Del      := Complex_Of((Cos(Angle)), (-(Sin(Angle))));

            if (Inverse_Transform) then
                    Del := Conjugate(Del);
            end if;
            Omega := Complex_Of(1.0,0.0);

            for IC in 1..C loop
                for IA in 1..A loop
                    for IB in 1..B loop
                        Sum := Data_Vector((((IA - 1)*C + (C-1)) * B) + IB);
                        for JCR in 2..C loop
                            JC := C_Plus_1 - JCR; - - No need to add C + 1 each time through loop
                            Sum := Data_Vector((((IA - 1)*C + (JC - 1))*B) + IB) + (Omega * Sum);
                        end loop; - - JCR
                        Counter := Counter + 1;
                        Temp(Counter) := Sum;
                    end loop; - - IB
                    Omega := Del * Omega;
                end loop; - - IA
            end loop; - - IC
            Data_Vector := Temp; - - assign output back to Data_Vector
        end Glassman;
------------------------------------------------------------------------------
    procedure FFT ( FFT_Data                    : in out Complex_Vector;
                    Inverse_Transform           : in boolean              ) is
            A   :   integer := 1;
            B   :   integer := FFT_Data'length;
            C   :   integer := 1;
        begin - - FFT
            while (B > 1) loop                  - - define the integers A, B, and C
                A := C * A;                     - - such that A*B*C = FFT_Data'length
                C := 2;
                while (B mod C) /= 0 loop
                    C := C + 1;
                end loop;
                B := B/C;                       - - B = 1 causes exit from while loop
                    Glassman (A,B,C, FFT_Data, Inverse_Transform);
                end loop;

                if Inverse_Transform then - - optional 1/N scaling for inverse transform only
                    for i in FFT_Data'range loop
                            FFT_Data(i) := FFT_Data(i) / float(FFT_Data'length);
                    end loop;
                end if;
        end FFT;
end FFT_Pack;
```

Figure 3.2. Ada Code for FFT Package

	No Output		Write to File	
	Average CPU Time in Seconds			
Points	Ada	FORTRAN	Ada	FORTRAN
500	0.1	0.1	0.1	0.3
1000	0.2	0.3	0.3	0.6
2000	0.4	0.6	0.8	1.3
3000	0.7	0.9	1.2	1.9
4000	0.9	1.2	1.7	2.6
5000	1.2	1.5	2.2	3.3
6000	1.5	1.8	2.7	3.8
7000	1.8	2.2	3.2	4.6
8000	2.0	2.5	3.6	5.2
9000	2.3	2.8	4.1	5.8
10000	2.6	3.2	4.6	6.5
11000	3.2	3.8	5.5	7.4
12000	3.1	3.8	5.6	7.8
13000	4.0	4.7	6.7	9.0
14000	3.9	4.7	6.9	9.3
15000	4.0	4.9	7.2	9.9

Table 3.1. Ada vs Fortran Execution Time Comparison

Although run times were roughly the same, the Ada compilation time was noticeably longer than that for FORTRAN. One reason for this extended compilation time is that Ada was developed as a strongly typed language, and performs numerous compilation checks to minimize run time errors. Ada also performs run time checks such as numeric range checking, which can be disabled using the *-S* command. Because Sun FORTRAN 3.1.1 does not perform this level of run time checking, the final Ada executable used in this comparison was compiled with run time checks disabled. A significant speed increase could be realized in the FORTRAN program by compiling with the *-fast* or *-fnonstd* commands, however, this compilation results in floating point outputs which do not conform to IEEE standards. Thus, neither the *-fast* FORTRAN nor the *-O* Ada optimization options were used during compilation. It is important to note that the accuracy of both programs outputs was comparable, since the outputs agreed down to the roundoff of the machine.

3.5 Summary

The FFT is the most involved algorithm in the GLOT. Coding this algorithm effectively in Ada provided confidence that the full transform could be implemented in Ada. The execution time comparison indicates that Ada is competitive with FORTRAN and is a viable language for other digital signal processing applications as well.

IV. Methodology

Programming a general N-point FFT routine was seen as the first priority software coding task, as discussed in Chapter 3. Before a full software implementation could be achieved, however, a prototype system was developed to test the theory of the GLOT. This chapter provides the link between theory and application, ranging from initial implementation of a prototype system to application of the GLOT to current signal processing problems. Personal Computer - Digital Signal Processing (PC-DSP), discussed below, was the first method used to test the GLOT. The program was used as a fast prototype, and aided in development and testing of the mathematical theory. The next test phase involved a computer program implementing the GLOT. The development of this program also relied heavily on the previous PC-DSP results. Some variations of the GLOT were implemented and tested, each working toward a different goal. Finally, the GLOT was applied to digital speech processing. The main focus of this segment of the research was to determine the minimum number of coefficients needed to reconstruct a quality reproduction of the input signal, while maintaining an acceptable computational penalty. The later portion of this chapter outlines the various techniques employed toward this end.

4.1 A Prototype System - PC-DSP

After carefully delineating the theory behind the GLOT, the first step in the evolutionary process of validating the theory was to develop a prototype system. The tool chosen for this purpose was Personal Computer - Digital Signal Processing (PC-DSP), an interactive menu-driven software package for use in common digital signal processing tasks (24). This program performs many of the functions required to test the GLOT, such as fast Fourier transforms (FFTs), point-by-point vector multiplications and additions, vector scaling, and the design and analysis of digital filters. Limitations which constrained the program included the inability to loop or perform an FFT on data of length other than 2^m, where m is an integer. Another disadvantage of the program was that each step had to be manually performed for each window of each data set. Some automation was available by using PC-DSP macro programs, where multiple PC-DSP commands could be loaded and executed from a file. These macro programs also had limitations, mainly involving insufficient memory for this application. In spite of these limitations, prototyping using PC-DSP provided the ability to determine the feasibility of the GLOT, and also limited the time required to code the eventual algorithm in a high-level language.

The signal shown in Figure 4.1 was chosen to test the analysis and synthesis algorithms. It consists of a sinusoid, a decaying exponential, a growing exponential, and another slowly varying sinusoid in the first through fourth partitions defined by the following equations:

$$s(x) = \begin{cases} sin(0.041x) & , \; 0 \leq x \leq 511 \\ 0.862exp[-0.006(x-512)] & , \; 512 \leq x \leq 1023 \\ 0.0402exp[0.006(x-1024)] & , \; 1024 \leq x \leq 1535 \\ 0.862 + 1.5sin[0.03(x-1536)] & , \; 1536 \leq x \leq 2048 \end{cases} \tag{4.1}$$

The window boundaries were chosen to be 512 and 1536, resulting in three analysis intervals of 512, 1024, and 512 points respectively. The window sizes were specifically chosen to be a power of two so that no zero-padding would be required to perform the subsequent FFT. Such padding would have created interpolation errors. In an effort to gain some insight into the effects of amount of window overlap on the reconstruction of the signal, the overlap at data sample 512 was 51 points (5% of the larger window) and the overlap at 1536 was 102 points (10% of the larger window).

The spectral coefficients obtained after performing the forward GLOT are given in Figures 4.2 through 4.4. Notice that only the first thirty coefficients of each window are shown. Although the transform yields as many coefficients as there are data points in the window, the coefficients beyond thirty for this signal were essentially zero. Only the first thirty coefficients were plotted to produce the coefficients of interest.

Intuitively, the spectrum plots are what would be expected for each analysis interval. The GLOT can be roughly equated to a Fourier transform, as both are based on representation of a signal using sinusoidal basis functions. Figure 4.2 has a strong $6th$ coefficient which results from the sinusoidal basis corresponding to the frequency of the sinusoid in

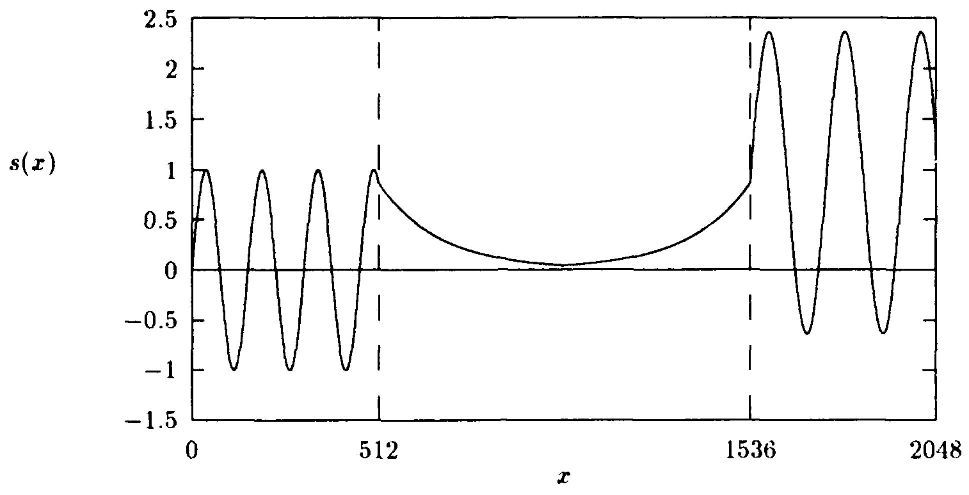

Figure 4.1. PC-DSP Input Signal

the first window. Because the frequency of this sinusoid and the nearest basis function frequency do not match exactly, other low frequency bases are also noticeable but of a lesser magnitude (see Section 1.1). Another source of noticeable low frequency spectral components in the first window is the overlapping window. A small portion of the low frequency exponential is characterized by the coefficients in the first window.

The signal in the middle segment of Figure 4.1 is slowly varying, and the corresponding spectrum (Figure 4.3) has many more significant coefficients than Figure 4.2. In general more spectral coefficients are required to represent a slowly varying time signal which is non-sinusoidal. By analogy, the Fourier transform of the unit gate or rect function is a sinc function (31:47,83), a spectral representation which also requires a significant number of damped oscillatory components.

As in Figure 4.2, Figure 4.4 shows a dominant coefficient (*4th*) corresponding to the frequency of the signal in the third interval. Additionally, the large *0th* coefficient corresponds to the DC offset in the interval.

After applying the inverse transform, the output shown in Figure 4.5 was achieved. The output signal and input signal are essentially identical, with an rms error of 2.47 x 10^{-5}.

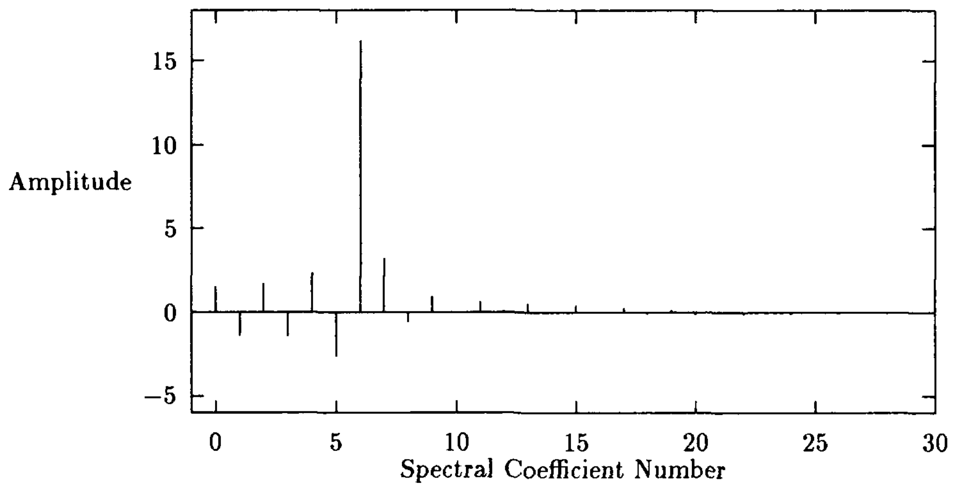

Figure 4.2. First 30 Spectral Coefficients of the First Window

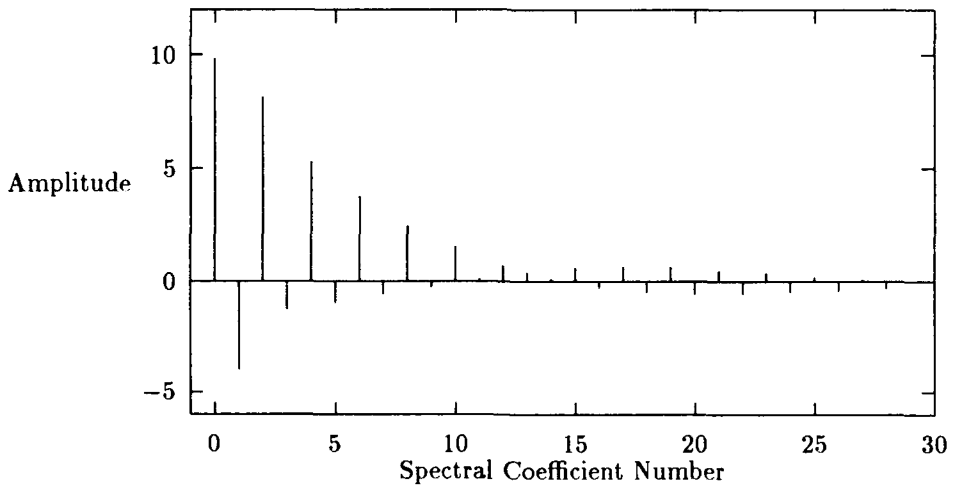

Figure 4.3. First 30 Spectral Coefficients of the Second Window

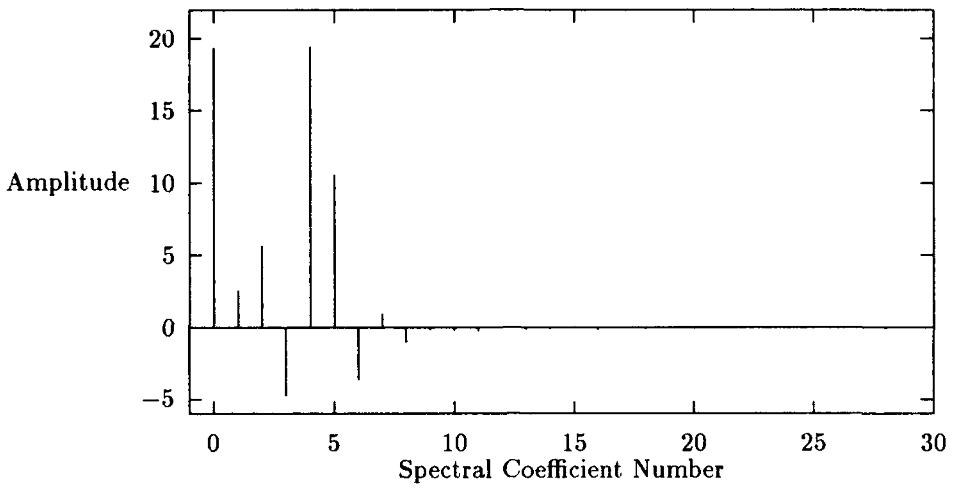

Figure 4.4. First 30 Spectral Coefficients of the Third Window

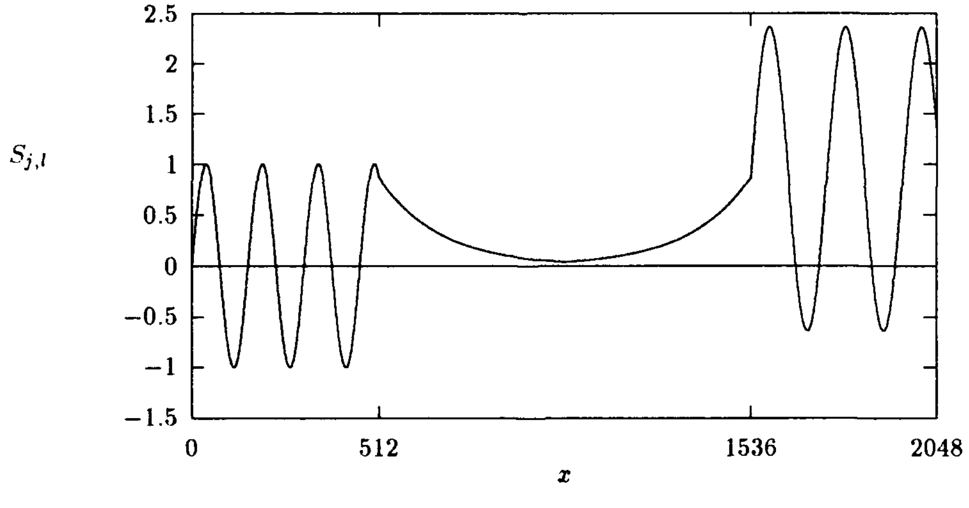

Figure 4.5. PC-DSP Reconstructed Signal

The use of PC-DSP was invaluable to the development of this thesis. It provided a proof of concept indicating the developed theory of Chapter 2 was correct and worthy of further research without fully coding the algorithm in a high-level language. Slight modifications and corrections to the theory were also facilitated, i.e. making inequalities in the derivation consistent for the "limiting" one point overlap case. Finally, a more retrospective benefit was that during coding of the algorithm, this same example was used. Having saved the plots after each step in the forward and inverse transform, the expected result was known and each step could be validated before proceeding to the next stage. Attempting to debug a program of this magnitude without knowing intermediate results would have been rather arduous.

4.2 Fast Fourier Transform

After developing the prototype system and validating the theory behind the GLOT, actual code was developed which could provide faster, more general results. The most important initial coding task was development of an FFT routine, and a full discussion of the Ada based FFT developed for this thesis is detailed in Chapter 3.

4.3 System Software Development

With a thoroughly tested FFT routine available, the software development unique to the GLOT could be initiated. One of the underlying goals during all software development was to ensure the code could be easily understood and modified. The use of Ada as a programming language aided in achieving this goal. Also, all algorithmic coding was made as general as was practical. This thesis is the incipiency of research regarding the GLOT, and it is essential that follow-on students understand the details of the software and can use it as a baseline for their subsequent work. Therefore, this section will briefly outline each package or subprocedure of the basic software program and how the package or subprocedure relates to the complete program. Further, each portion of the code will be referenced to the appropriate section or equation in Chapter 2. A complete listing of all source code (with similar referencing) is provided in Appendix A.

4.3.1 Main Program Subprocedures

4.3.1.1 Get_File_Data Procedure *Get_File_Data* prompts the user for the name of the data file which contains all the information necessary to run the program. This input file must include the following entries :

1. The file containing the data set to be processed
2. The desired filename for the derivative data
3. The desired filename for the spectral coefficient data
4. The desired filename for the output data
5. The desired filename for the error data
6. The desired filename for the information pertaining to this run
7. The number of points to be processed
8. The number of data partitions
9. The partition point and overlap size for each partition

Three different stages in the transformation were of interest, including the derivative of the spectral coefficients (used mainly in initial debugging), the spectral coefficients, and the output (or reconstructed signal). This data was stored in the user defined files specified in entries 2-4 above. The point-by-point error between the output and input is captured in the file specified by entry 5. General information pertaining to the current data run and any compression or filtering employed can be written to the information file specified by entry 6. After the number of partitions is entered, the partition point and number of points overlap is loaded for each partition. Also in this subprocedure, the first window of input data is copied to the end of the file. This allows so the data to be processed as a periodic waveform in an effort to avoid boundary noise at the front and back of the reconstructed signal. An example input file is given in Figure 4.6. This particular input file specifies that 1000 samples will be extracted from file "input.dat". The results of interest will be written to the appropriate filename. The data will be partitioned into 4 segments, with overlap of 10 points for the first two intervals, and 20 points for the last two intervals.

```
input.dat
derivative.dat
alpha.dat
output.dat
error.dat
info.dat
1000
4
200
10
400
10
600
20
800
20
```

Figure 4.6. Example Input File

4.3.1.2 Multiply_By_Window Procedure *Multiply_By_Window* creates a window (defined by equation 2.8) based on the size of the data to be processed and the overlap on either side and multiplies this window by the data. It also folds in the appropriate points to create a vector from a_j to a_{j+1} as defined in equation 2.9.

4.3.1.3 Compute_Coefficients Procedure *Compute_Coefficients* performs an even extension of the data, and an FFT as defined in steps 5 and 6 of Section 2.1.2.4. Note here that the points into the routine are from a_j to a_{j+1} or $N + 1$ points, while the coefficients out of the routine are from a_j to $a_{j+1} - 1$ or N coefficients.

4.3.1.4 Reconstruction_FFT Procedure *Reconstruction_FFT* is the first step in the reconstruction of the signal, involving an odd extension of the data and an FFT as defined in steps 1 and 2 of Section 2.1.3. The output data is the imaginary output from the FFT.

4.3.1.5 Divide_By_Windows Procedure *Divide_By_Windows* employs the window created in *Multiply_By_Window* to reconstruct the signal as defined in step 3 of Section 2.1.3. The output data from $a_{j+1} - \epsilon_{j+1}$ to a_{j+1} cannot be reconstructed during the current window processing loop because this segment requires the processing of data from a_j to $a_j + \epsilon_j$ of the next window. Therefore, during any given window processing loop, output data from $a_j - \epsilon_j$ to $a_{j+1} - \epsilon_{j+1}$ is calculated and written to the output file.

4.3.1.6 Do_Work Procedure *Do_Work* performs the complete GLOT by calling the procedures discussed above. Each data segment is processed in the main loop and the variables characterizing the current data segment are initialized as a first step inside this loop. Any processing of the spectral coefficients to obtain signal compression can be implemented between procedures *Compute_Coefficients* and *Reconstruction_IFT*.

This procedure also implements the solution to the finite signal length problem outlined in Section 2.2. The first and last data segments are handled as special cases. The reason for this special processing is the absence of data left of the first segment and right of the last segment. Lack of data in these regions means non-overlapped processing on $[a_j, a_j + \epsilon_j]$ in the first segment, and on $[a_{j+1} - \epsilon_{j+1}, a_{j+1}]$ in the last, leading to edge effects in the reconstructed output. The problem is handled by processing the first data segment twice. Looking back to procedure *Get_Data* discussed in Section 4.3.1.1, the first window of data was written to the end of the input vector. When the input data is passed to *Do_Work* for processing, the first window of data will be processed once during the first processing cycle ($j = 0$) and once during the last (j =Partitions'last-1). The first time through the processing loop, there will be perfect reconstruction in (1) the region of no overlap $[a_j + \epsilon_j, a_{j+1} - \epsilon_{j+1}]$, and (2) the overlapped region with the right adjacent interval $[a_{j+1} - \epsilon_{j+1}, a_{j+1}]$. The reconstructed values from $[a_j, a_j + \epsilon_j]$ will be determined in the final loop, where the left adjacent window of data is the last input segment. These values are written to the beginning of the output file in the loop following a call to procedure *Divide_By_Windows* (see Appendix A). The last input data segment is properly processed during the second-to-last processing loop, since the first data segment has been appended to the right of the end data. By "wrapping" the data in this manner, the reconstructed output for the first and last data segments is free from edge effects caused by non-overlapping intervals. All other segments of data are inherently immune to this effect as discussed in the theory of Chapter 2.

4.3.2 Package Type_Package Package *Type_Package* contains all types used in the program, and these types are global to all packages and subprocedures. Unconstrained arrays are used wherever possible to allow dynamic memory allocation. Types Direction, Phase_Vector, and Phase_Type were used as part of compression techniques which will be discussed in subsequent sections.

4.3.3 Package FFT_Pack Package *FFT_Pack* is fully explained and documented in Chapter 3.

4.3.4 Package Vector_Package Package *Vector_Package* contains three functions which contributed to the readability of the main program, and operated on unconstrained array sizes. Function *Even_Extend* implemented the even extension defined in Section 2.1.2.4, step 5, and is called from procedure *Compute_Coefficients* within the main program. Function *Odd_Extend* implemented the odd extension defined in Section 2.1.3, step 1, and is called from procedure *Reconstruction_FFT*. Finally, function *Reverse_Assignment* was used to assign window values to the left window segment from the right window segment within procedure *Divide_By_Windows.* This function exploits the fact that overlapping window segments must be symmetric about a_j. Without this function, an additional window vector would have had to be passed out of procedure *Multiply_By_Windows* and into procedure *Divide_ByWindows.* This package can be expanded to include future vector manipulation requirements, or could be linked for use by another algorithm requiring these functions.

4.3.5 Package Complex_Pkg Package *Complex_Pkg* contains basic complex number manipulations on real and complex numbers and vectors.

4.3.6 Independent Interval Processing All of the software code referenced thus far was based on overlapped adjacent intervals, except for the two intervals at the signal boundaries. These two intervals were treated as special cases as discussed in Section 4.3.1.6.

Applying the concept of independent interval processing discussed in Section 2.3 required significant modifications to this software program. All affected procedures are given in Appendix C and replace those given in Appendix A. The most important distinction evident in the changes is found in Procedure *Do_Work*. Previously, reconstruction of the signal in the jth interval on $[a_{j+1} - \epsilon_{j+1}, a_{j+1}]$ was delayed until the processing of the next interval. This requirement is based on the dependency between the overlapped region of adjacent signal intervals. Using the approach of Section 2.3, however, each interval can be fully reconstructed independent of any other intervals. The independent processing of data can be an advantage in terms of real-time processing, error control, parallel processing, and decreased buffer requirements, all of which will be discussed in Chapter 5.

4.4 Applications

After developing a software program which implemented the GLOT, the program was used to investigate various potential applications. The research discussed in this chapter concerning speech involves various attempts to efficiently encode the decomposed spectral coefficients in a manner that allowed for a quality reconstruction. Applying the GLOT to speech signals was a natural extension of previous work in this area (18:975-976). Further, it was assumed that speech, being sinusoidal in nature (32), would lend itself to a transform based on sinusoidal representation. The following sections detail initial testing and evaluation of the algorithm, including its use in solving the finite length data problem, perfectly reconstructing independently processed intervals, and compressing speech.

4.5 Initial Testing

The PC-DSP example outlined in Section 4.1 was used throughout development of the code outlined in Section 4.3. After ensuring the signal defined in equation 4.1 could be reconstructed to roundoff error, the program needed to be tested with an actual signal. A sample speech signal from a professional speech data base (22) was used, which consisted of the phrase "American newspaper reviewers like to call his place nihilistic", spoken by a male speaker. This 60,080 sample speech file was sampled at 16 kHz and represented using 16 bit integers. These numbers become important when discussing data rates and

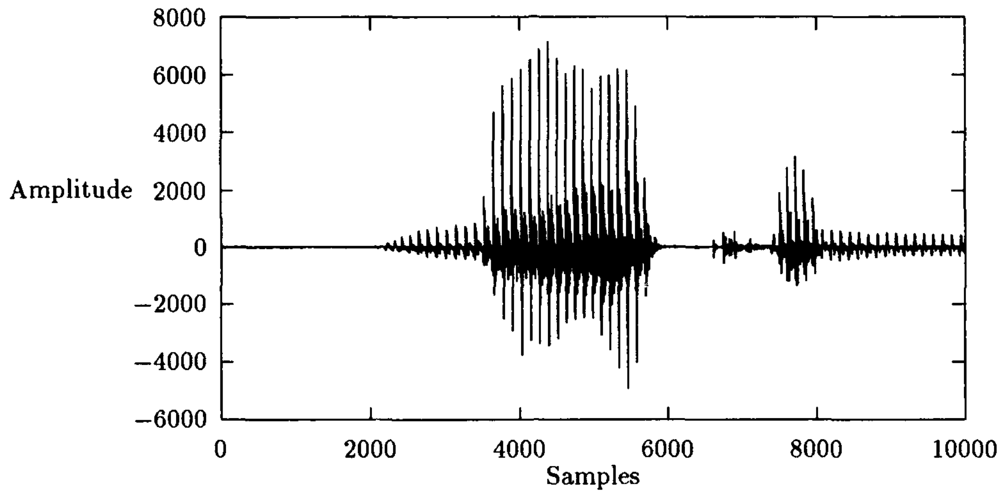

Figure 4.7. Sample Speech Signal

compression techniques. Figure 4.7 displays the first 10,000 samples of the data which correspond roughly to the word "American...". The signal was divided into equal sized windows of 512 points with a one point overlap. The one point overlap was chosen to determine an upper bound on the amount of edge effects as a function of window overlap. This is the least amount of overlap allowed for the GLOT.

An example of the spectral coefficients calculated in the forward transform is given in Figure 4.8. The coefficients in this interval correspond to input samples from 4096 to 4608. Figure 4.7 demonstrates that this interval contains significant speech data, yet our spectral coefficients seem to be insignificant in the high frequency range. Although the exact nature of the response of the system to eliminating coefficients is unknown until tested, this initial plot lends confidence to the idea that a significant amount of compression may be possible using the GLOT, without significant degradation of the signal and without extremely burdensome computational complexity. First, however, it was necessary to determine the ability of the algorithm to reconstruct the signal using all spectral components.

The signal was reconstructed using all of the spectral coefficients and the result is given in Figure 4.9. If this plot looks quite similar to that in Figure 4.7, it should. They are

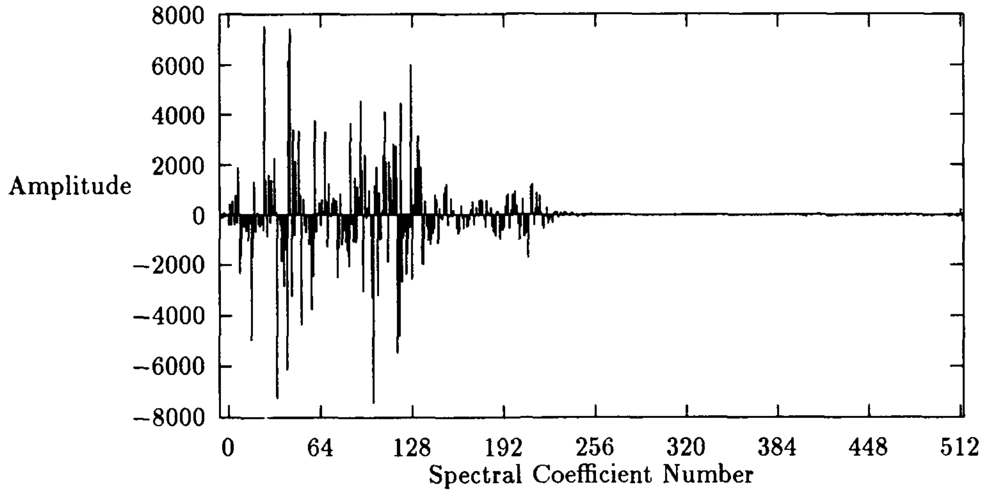

Figure 4.8. Sample Spectral Coefficients in One Window on Interval [4096, 4608]

exactly the same. The ESPS program used to generate recordings required integer inputs; the output signal, therefore, was rounded to the nearest integer before being sent to a file. The input signal also consisted of integers. After reconstructing this speech signal using all the coefficients and a one point overlap, each of the 60,080 samples of the output and input were exactly equivalent.

The perfect reconstruction of the first and last data segments, using the method outlined in Section 2.2, was also confirmed. Using the GLOT in this manner provides for a solution to finite length signal processing previously unavailable. This problem is of significance in applications such as image processing, and of lesser importance in speech processing where data flow is continuous. This result demonstrates the potential application of the GLOT to a wide range of signal processing problems beyond the speech application discussed later in this section. The three goals of a nonexpansive transform, perfect reconstruction, and no additional complexity were all met using this approach.

As a separate effort, the independent interval processing program detailed in Appendix C was tested. This program provided perfect reconstruction of the input signal using 512 sample intervals and a 10% overlap. As this concept was developed late in the

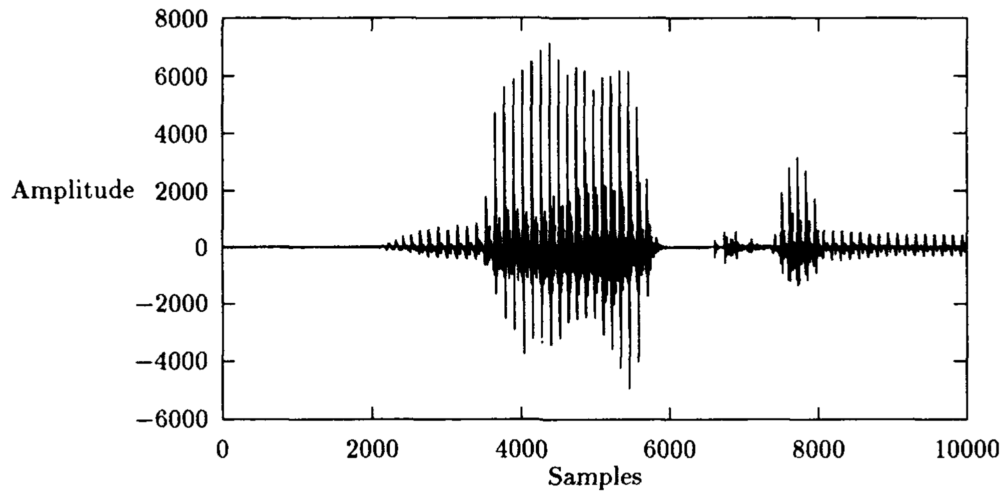

Figure 4.9. Reconstructed Speech Signal

thesis effort, no further experiments were performed using this program. Chapter 5 discusses the potential for follow-on research in the area of independent interval processing using the GLOT.

Obtaining perfect reconstruction with a one point overlap was a very significant result. The previous work in this area, in particular by Malvar (21), (18) and Princen and Bradley (27), were based on a 50% overlap which requires more processing since each point is processed as part of two intervals. Further, Malvar's fast algorithm approximates the LOT (21:556-557), while the GLOT requires no approximations. Since the GLOT is of the same complexity as the FFT, it is comparable to Malvar's algorithm (21:553). This result also formed a baseline from which compression techniques could be investigated in an attempt to acquire the basic elements of a low bit rate speech encoder. As discussed in Section 1.2, perfect reconstruction, when using all coefficients, is an essential requirement to further pursuing the merits of a specific decomposition/reconstruction algorithm. Because of the excellent results with a one point overlap, subsequent efforts focussing on compression techniques were primarily based on this minimal overlap.

4.6 Exploring Various Compression Techniques

Building on the success of the previous section, various compression techniques could be investigated in an effort to reduce the number of bits required to represent a speech signal with a high degree of accuracy. One problem associated with determining the most promising approaches to compressing speech is that success is defined by the ear. Firm quantitative measures for speech coding success have yet to be developed, and subjective measures are often employed to quantify a degree of success. Ongoing research attempts to link objective measures with the subjective success or failure of various compression techniques (see, for example, Quackenbush *et al.* (28)). No firm guidelines could be used to determine which approaches would be most promising, because the approach best suited to any algorithm when dealing with speech signals is idiosyncratic to the method employed (15). The following sections outline the compression techniques investigated pursuant to this thesis. Each of these techniques required some processing in addition to that accounted for in Appendix A. Additional code used in conjunction with the following techniques will be provided as part of the appropriate section.

4.6.1 Absolute Spectral Thresholding Absolute Thresholding was one of the most basic and effective techniques employed in reducing the number of spectral coefficients, while maintaining a high degree of speech quality. The code given in Figure 4.10 was added to Procedure *Do Work* following the procedure call to *Compute Coefficients* and replaces the print loop at that location.

As noted in Section 1.4, the three goals of a high quality, high compression, low complexity coder must all be weighed. This technique has a relatively low complexity, since only a simple threshold comparison is required beyond the basic transform. While setting a threshold is a simple technique, the choice of a proper threshold is arbitrary. In general, the proper threshold will depend on the data being processed and effect the compression capability of the system. With this in mind, testing was accomplished to determine the quality versus compression trade-off. Four different compression ratios were used to process the test phrase, corresponding to 85.0%, 92.5%, 96.2%, and 98.1%. This means that the stated percentage of least significant coefficients were not used, or "zeroed

```
for x in Start_Data..(End_Data-1) loop
    if abs(Transformed_Data(x)) < Alpha_Threshold then
        Transformed_Data(x) := 0.0;
        if j < (Partitions'last - 1) then
            Alpha_Compression_Counter := Alpha_Compression_Counter + 1;
        end if;
    end if;

    Float_Io.put(Alphafile,Transformed_Data(x));
    Text_Io.new_line(Alphafile);
end loop;
```

Figure 4.10. Additional Code for the Absolute Threshold Compression Technique

out" prior to reconstruction of the signal. The following assumptions allowed for a bit rate calculation corresponding to the percentages chosen above :

1. Downsampling the input from 16 kHz to 8 kHz would not significantly affect the results.
2. The spectral coefficients could be represented with 4 bits.
3. Additional overhead bits need not be accounted for.

Bit rates were then computed by :

$$R_b(\frac{\text{bits}}{\text{sec}}) = 8000(\frac{\text{data sample}}{\text{sec}}) * 1(\frac{\text{coefficient}}{\text{data sample}}) * 4(\frac{\text{bits}}{\text{coefficient}}) * (1 - C). \qquad (4.2)$$

where R_b is the bit rate of the system, and C is the compression ratio of eliminated coefficients. This formula yields bit rates of $4800, 2400, 1200$, and 600 bits/sec when using the compression ratios mentioned above. These bit rates are commonly used in speech processing systems. The same data rates were tested here to provide a bit rate comparison with existing techniques.

The thresholded coefficients of one interval for compression percentages of 85.0% and 98.1% are shown in Figure 4.11 and Figure 4.12, respectively. Notice the significant reduction in the number of non-zeroed coefficients, especially in Figure 4.12, when compared to the unthresholded coefficients in Figure 4.8. The first $10,000$ points of the output wave-

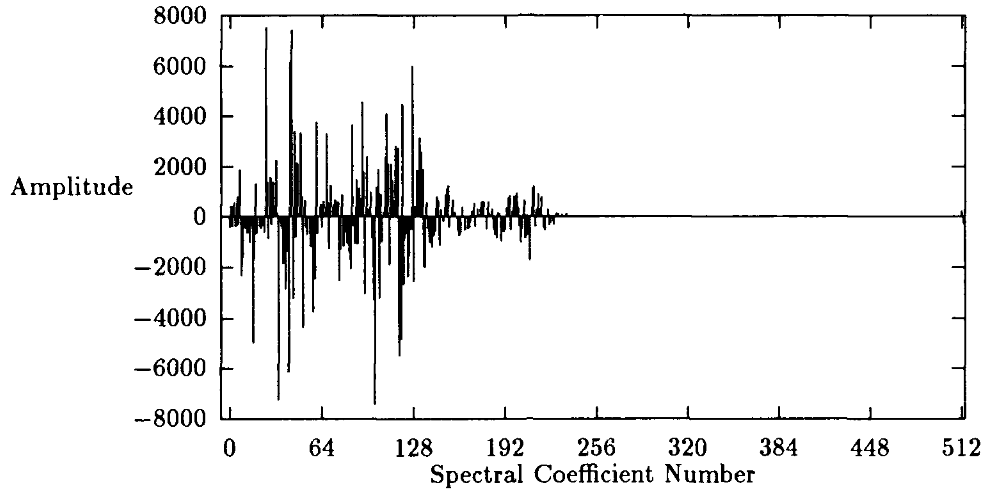

Figure 4.11. Absolute Threshold Spectral Coefficients (85.0%) on Interval [4096, 4608]

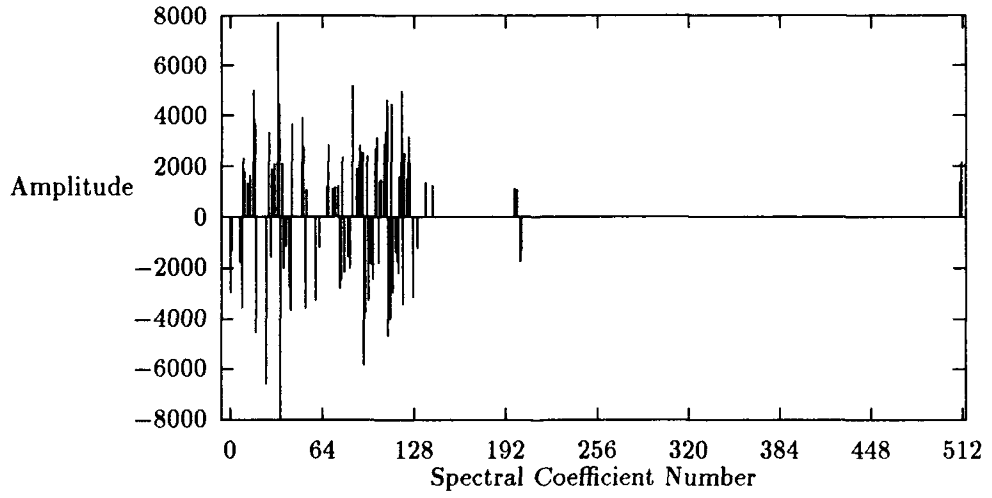

Figure 4.12. Absolute Threshold Spectral Coefficients (98.1%) on Interval [4096, 4608]

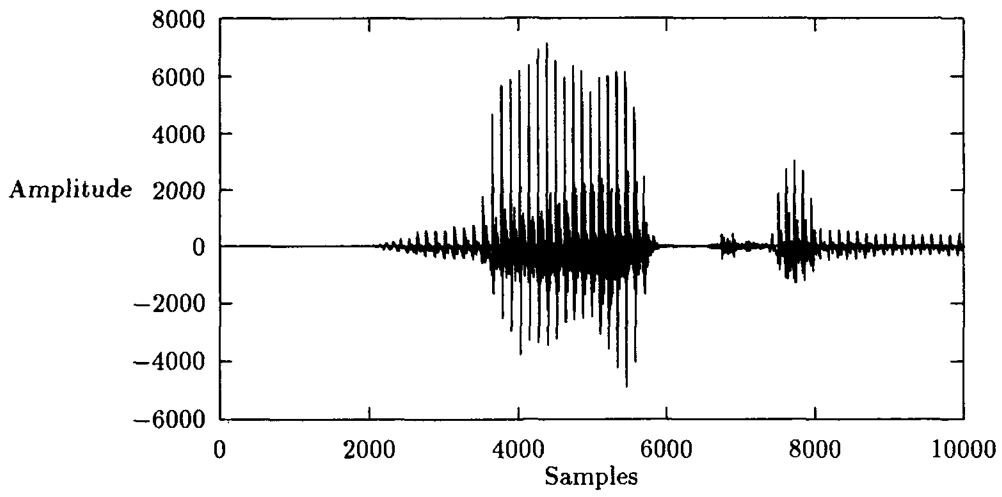

Figure 4.13. Absolute Threshold Output Waveform with 85.0% Compression

form for these same compression percentages are given in Figures 4.13 and 4.14. When compared to Figure 4.9, Figure 4.13 provides a better match to the perfectly reconstructed waveform than Figure 4.14, especially when the speech is of low amplitude. Of course, the final test of the success or failure of these outputs in matching the input waveform depends on the ear. After listening to the absolute thresholded outputs, some interesting observations were made. The reconstructed signal at an 85.0% threshold level was of high quality. In each recording, noise was evident in the form of "pinging" or extraneous tones. The speaker's voice, however, especially at 85.0% thresholding, was not distorted. In other words, the speaker, as well as the speech, could be discerned in the recording. In addition to the background tones, the other form of noise was the elimination of parts of speech near word boundaries. This can be seen in Figure 4.14 as the degraded reconstruction just after sample 2000 and again between sample 6000 and 8000. Even at the 85.0% level, the final syllable of the final word "nihilistic" was not discernible, but the problem was most significant at higher compression levels. The 98.1% thresholded output was not recognizable, due to both forms of noise discussed above.

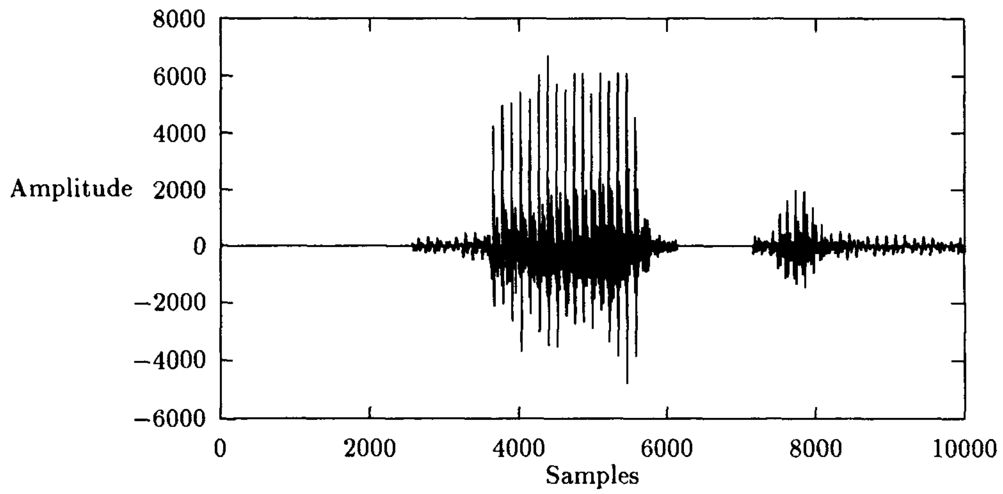

Figure 4.14. Absolute Threshold Output Waveform with 98.1% Compression

A modification to this approach was also implemented after viewing the results of the absolute threshold technique. At high compression ratios, a single (or small number of) remaining coefficient(s) might result in a reconstruction containing sinusoidal noise in the entire interval. Figure 4.15 is an example of two such intervals. In the two adjacent intervals shown, only one coefficient was of large enough magnitude to survive the thresholding. The resulting reconstruction is a low frequency sinusoid from the low frequency coefficient in the left interval, and a high frequency sinusoid from the high frequency coefficient in the right interval. Figure 4.16 yields this reconstruction, which is a form of noise in the output waveform. (Note also the 180 deg phase shift in the low frequency sinusoid resulting from a negative coefficient, as expected). To lessen this problem, and allow a more judicious choice of important coefficients, the following rule was applied : any coefficient above the threshold, but with neither adjacent coefficient above the threshold, will be zeroed as if it had not met the threshold criterion. Besides the visible sinusoidal noise, an underlying assumption in making this rule was that coefficients containing a significant amount of signal information are generally grouped together. This assumption was based on viewing numerous spectral plots. Based on this assumption, the converse of

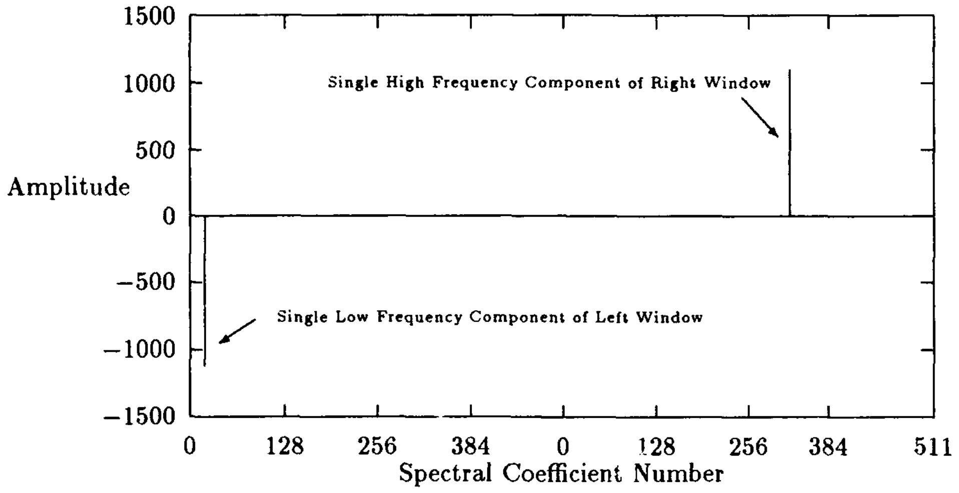

Figure 4.15. Single Spectral Coefficient in Each of Two Adjacent Intervals

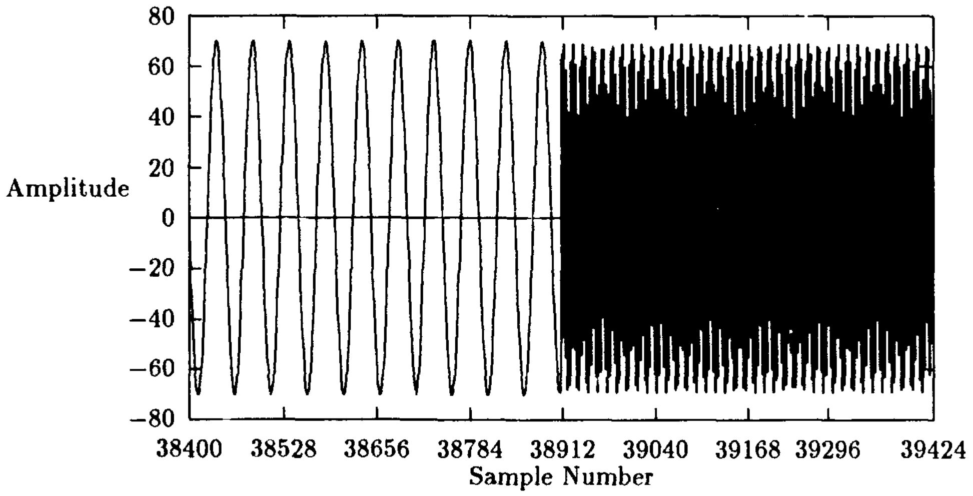

Figure 4.16. Reconstructed Data in Two Adjacent Windows, [38400 – 39424]

the aforementioned rule was also applied : any coefficient below the threshold, but with both adjacent coefficients above the threshold, will be retained for reconstruction. The code for this modified absolute threshold method is given in Figure 4.17. Recordings of the output waveforms using this modified absolute threshold approach were indistinguishable from the recordings when using the original approach. The comparison was performed for each of the four compression percentages. Some differences in the waveform were apparent, but they were not significant enough to be distinguished by the ear.

4.6.2 Relative Spectral Thresholding The next approach was similar to the absolute threshold method, but each coefficient was scaled prior to comparison with the threshold. The scale factor was simply the sum of all coefficients in the window, or the integral of coefficients in the window. Adding this parameter into the calculations was an attempt at providing a better reconstruction for intervals containing edges of words or near speech/non-speech boundaries. This portion of the reconstructed signal was degraded substantially using the absolute threshold method, especially at higher compression percentages (or lower bit rates), as discussed in Section 4.6.1. The code for this new approach is given in Figure 4.18. Figures 4.19 and 4.20 are the reconstructed waveform after applying the relative threshold technique. The 85.0% reconstruction has some spikes in the output at various sample locations. At the 93.1% level, the output is severely distorted in many intervals. When comparing a recording of these outputs to the absolute threshold technique for similar compression percentages, the absolute threshold method produced a better quality reproduction. There was more background noise or "tones" in the sound of the relative thresholded reproduction, and the sound of the voice also changed slightly (affecting the recognition). The 93.1% compressed reconstruction was mostly pure noise to the ear, as would be expected from viewing Figure 4.20. At 85.0% compression, the relative threshold signal, while worse than its absolute threshold equivalent in general, had one redeeming quality. The signal at word boundaries was often reproduced using the relative threshold technique whereas it was eliminated in the absolute threshold output. The relative threshold method reproduced the "...tic" of "nihilistic", discussed in Section 4.6.1. This advantage is the result of forcing coefficients from each window to be retained, thereby allowing a reproduction even where there is minimal signal energy. The

```
- -Take care of first coefficient separately
- -so the first time through the following
- -loop there is a (x - 1) point to look at
if abs(Transformed_Data(Start_Data)) < Alpha_Threshold then
   Transformed_Data(Start_Data) := 0.0;
   if j < (Partitions'last -1) then
      Alpha_Compression_Counter := Alpha_Compression_Counter + 1;
   end if;
end if;
Float_Io.put(Alphafile,Transformed_Data(Start_Data));
Text_Io.New_Line(Alphafile);

for x in (Start_Data+1)..(End_Data-1) loop
   if abs(Transformed_Data(x)) < Alpha_Threshold then
      if Transformed_Data(x-1) = 0.0 then
         Transformed_Data(x) := 0.0;
         if j< (Partitions'last -1) then
            Alpha_Compression_Counter := Alpha_Compression_Counter + 1;
         end if;
      elsif abs(Transformed_Data(x+1)) < Alpha_Threshold then
         Transformed_Data(x) := 0.0;
         if j < (Partitions'last -1) then
            Alpha_Compression_Counter := Alpha_Compression_Counter + 1;
         end if;
      end if;
   elsif Transformed_Data(x-1) = 0.0 then
      if abs(Transformed_Data(x+1)) < Alpha_Threshold then
         Transformed_Data(x) := 0.0;
         if j < (Partitions'last -1) then
            Alpha_Compression_Counter := Alpha_Compression_Counter + 1;
         end if;
      end if;
   end if;
   Float_Io.put(Alphafile,Transformed_Data(x));
   Text_Io.New_Line(Alphafile);
end loop;
```

Figure 4.17. Additional Code for the Modified Absolute Threshold Compression Technique

```
Alpha_Sum := 0.0;
    -- Alpha_Sum is recomputed for each interval

for x in Start_Data..(End_Data-1) loop
    Alpha_Sum := Transformed_Data(x) + Alpha_Sum;
end loop;

for x in Start_Data..(End_Data-1) loop
    if (Transformed_Data(x)/Alpha_Sum) < Rel_Threshold then
      Transformed_Data(x) := 0.0;
      if j< (Partitions'last -1) then
        Alpha_Compression_Counter := Alpha_Compression_Counter + 1;
      end if;
    end if;

    Float_Io.Put(Alphafile,Transformed_Data(x));
    Text_Io.New_Line(Alphafile);
end loop;
```

Figure 4.18. Additional Code for the Relative Thresholding Compression Technique

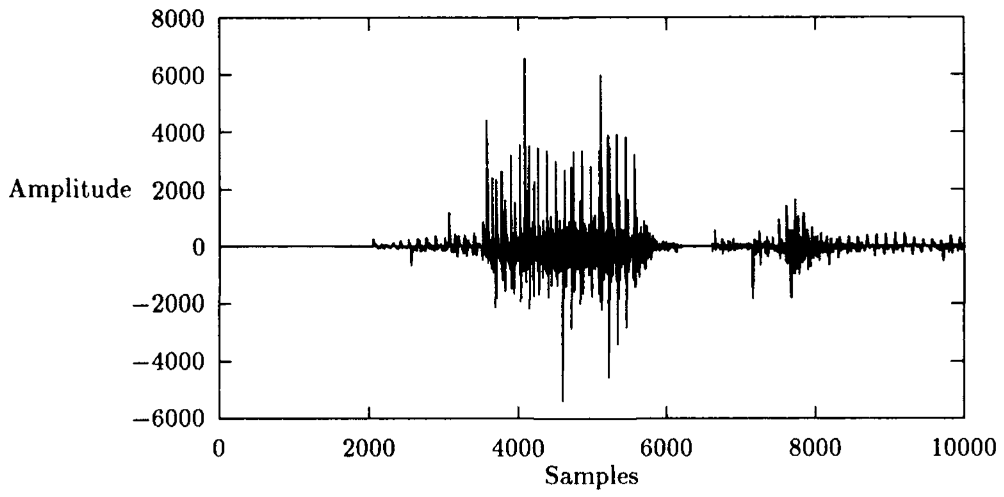

Figure 4.19. Relative Threshold Output Waveform with 85.0% Compression

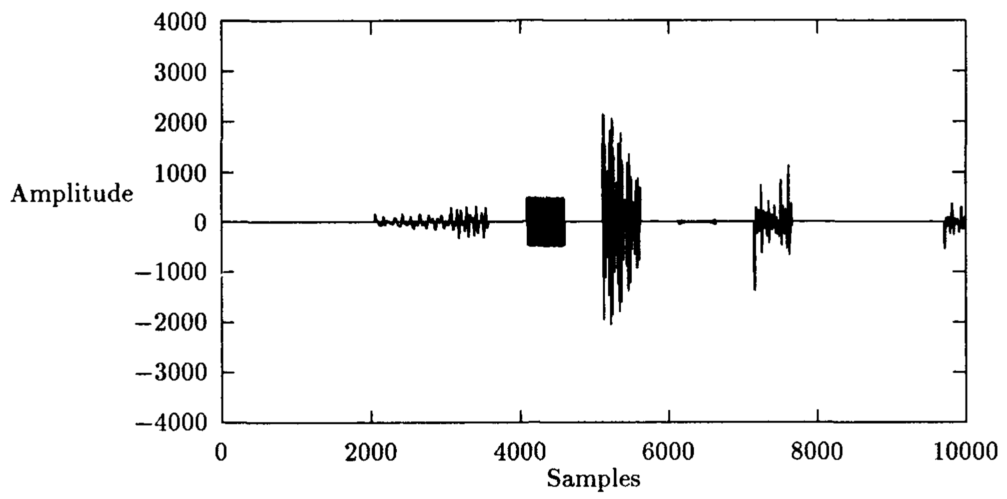

Figure 4.20. Relative Threshold Output Waveform with 98.1% Compression

tradeoff was that coefficients were removed from the windows containing a strong signal, creating excess noise in numerous intervals. This noise overshadowed any benefit resulting from use of the relative threshold technique.

4.6.3 Polynomial Fitting of the Spectrum All of the speech compression techniques discussed thus far had unique merits and shortcomings, but none was far superior in performance to the others. The simplest technique, the basic absolute threshold, provided the best reconstruction, yet none of the others were significantly more complex. The amount of required coefficient processing was an important parameter to minimize. For this reason, the initial methods tested were the least complex. It became apparent, however, that further reductions in the number of coefficients required for high quality speech reconstruction would require more complicated spectral processing. Viewing the unthresholded coefficients, it appeared is if they might be modelled by a polynomial curve. The following rules were established as a criteria for what information would be extracted from the coefficients and used in the reconstruction :

1. A noise threshold will be established below which no coefficients will be kept.

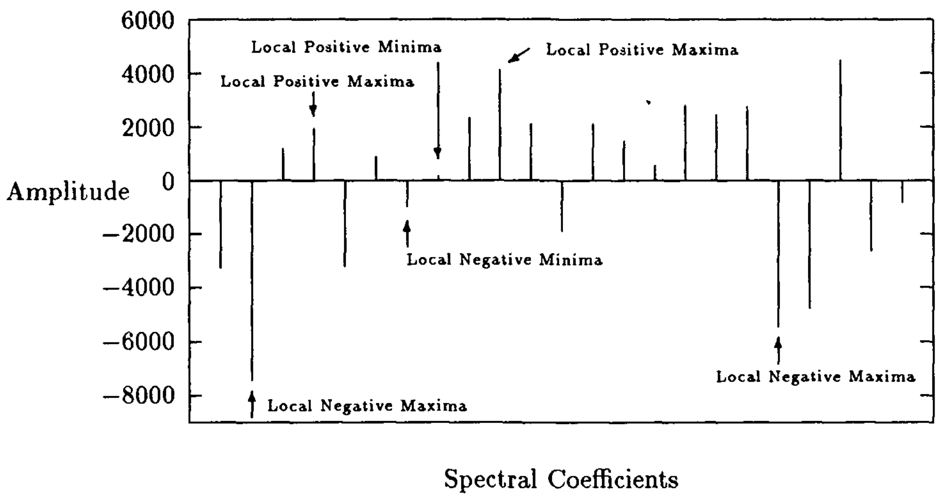

Figure 4.21. Sample Spectral Coefficients Tagged for Polynomial Fit Processing

2. From the remaining coefficients, the local maxima and minima for *both* the positive and negative coefficients will be retained.

3. Of those coefficients above the noise threshold, but not maintained as a maxima or minima, the phase will be maintained.

Figure 4.21 is a portion of the coefficients given by Figure 4.8. The first three maxima or minima which would be saved in this interval based on the above rules are labeled appropriately. The noise threshold was assumed to be below any of the coefficients in this figure. The following rules governed reconstruction of the signal given this saved information :

1. A deterministic polynomial curve will be fitted using the strictly positive or strictly negative local maxima, local minima, and local maxima (see Figure 4.21) as the left, middle, and right points on the curve respectively. This process will result in both a positive and negative polynomial curve being defined at any given point in the spectrum.

2. Any coefficient which was not a maxima or minima, but was above the noise threshold, should be estimated by the value of the appropriately phased polynomial curve defined at that location.

3. Any coefficient which was below the noise threshold will be assumed to be zero.

This method required the addition of three new procedures outlined in Appendix B. Procedure *Maxima_Minima* extracted the positive and negative maxima and minima, the required phase information, and some initial conditions for use in the reconstruction. Procedure *Polynomial_Fit* defined the parameters to be used in procedure *Fit_Slice*, which computed the actual polynomial and reconstructs the coefficients. At this point, the basic reconstruction is performed on the estimated coefficients.

This method is different from any of the other methods in that the amount of possible compression is set by the data rather than varied based on a compression threshold. Although some range of compression can be achieved by modifying the noise threshold, the compression percentage is essentially the ratio of relative maxima and minima spectral coefficients to data samples. An advantage of this technique is that the choice of threshold is not arbitrary, but related only to the amount of noise in the signal. Viewing the data in non-speech intervals, it was determined that the noise reached an amplitude level of approximately 35. Setting the threshold to this level resulted in a compression ratio of 20%. A threshold of 60 yielded a compression ratio of 85.0%, analogous to data accumulated for previous methods. Varying the threshold from 35 to 60 did not perceptibly alter either the viewed or recorded reconstructed output. The coefficients remaining after extracting the maxima and minima for one interval are given in Figure 4.22. The coefficients in this interval are much more sparse than the original unthresholded coefficients given in Figure 4.8. The reconstructed output with 85.0% compression using the polynomial fit technique is given in Figure 4.23.

Figures 4.24 through 4.26 are provided to demonstrate the most important aspect of the polynomial fit technique, and the step which creates noise in the reconstructed output. Figure 4.24 is a plot of unthresholded spectral coefficients 64 – 128 from the interval analyzed in Figure 4.8. Figure 4.25 are the remaining (positive and negative) maxima and

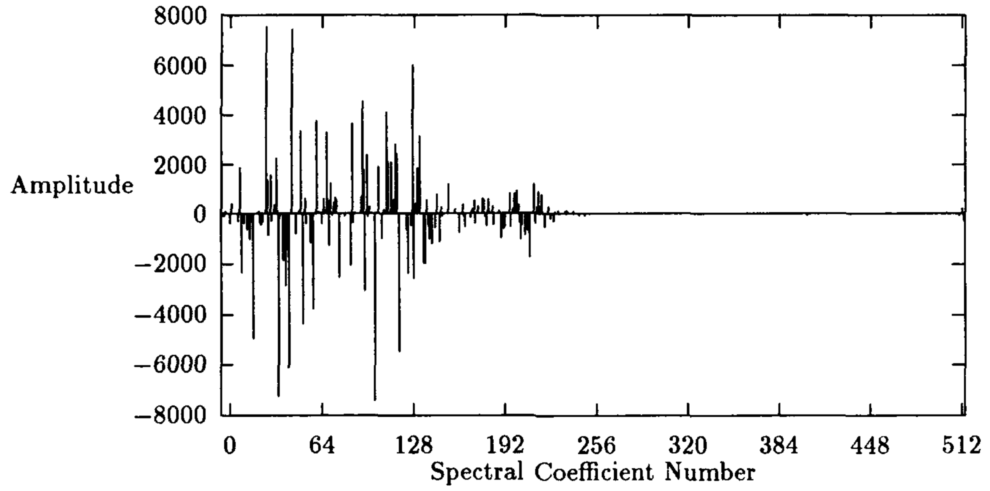

Figure 4.22. Polynomial Fit Spectral Coefficients (85.0%) on Interval [4096, 4608]

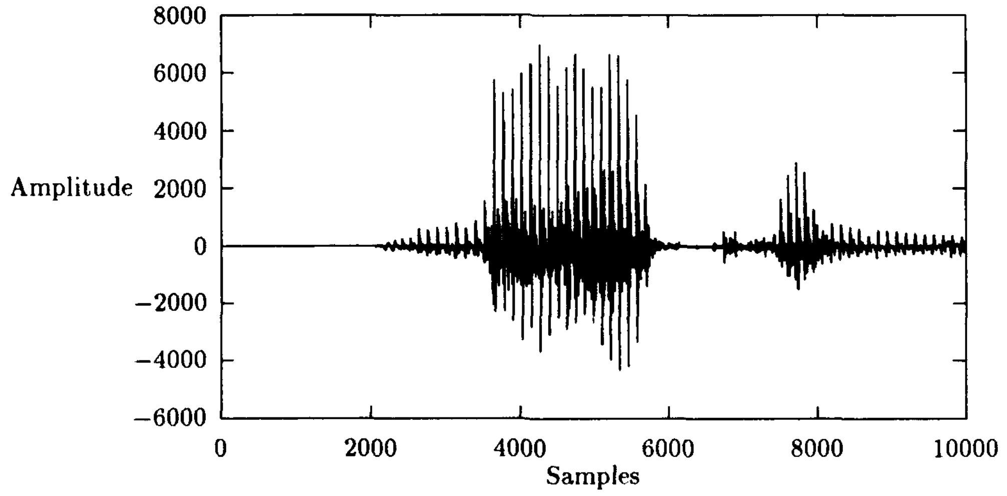

Figure 4.23. Polynomial Fit Output Waveform with 85.0% Compression

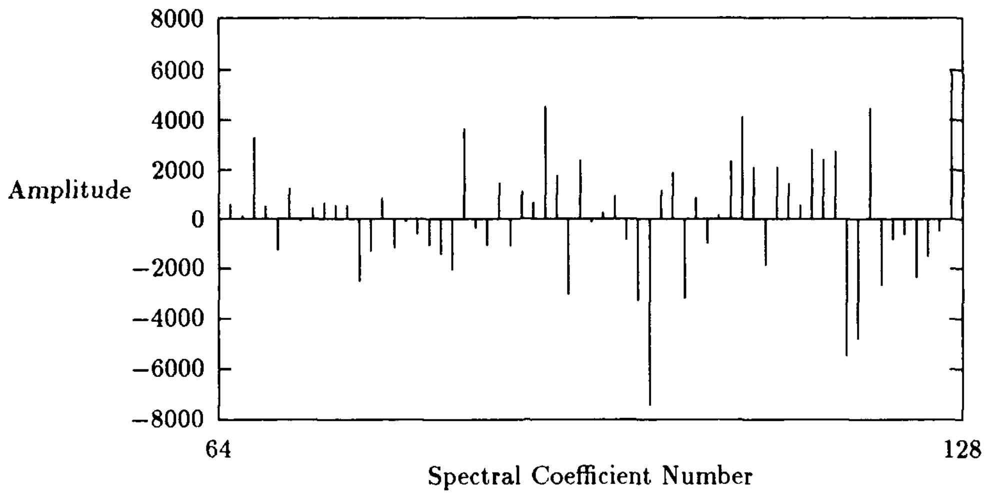

Figure 4.24. Unthresholded Coefficients 64 – 128 of Figure 4.8

minima which will be used to reconstruct the signal, along with a sign vector. Figure 4.26 gives the reconstructed or estimated spectral coefficients based on the polynomial fit criteria discussed above. Comparing Figures 4.24 and 4.26, it is apparent that in some regions, the polynomial fit accurately estimates the actual coefficients. In other regions, however, the coefficients cannot be modelled accurately with a polynomial curve. Reconstructing the signal using coefficients which have not been accurately modelled creates noise in the output waveform. The full extent of this noise can only be ascertained, however, by listening to the recorded output.

The polynomial fit output was of the same audio quality as the absolute threshold output of similar compression ratio. It also had the advantage, as in the case of the relative threshold technique, of retaining some speech sounds at word boundaries which were eliminated in the absolute threshold output. In this sense, the polynomial fit technique was marginally superior in audio reproduction, at the expense of some additional computations.

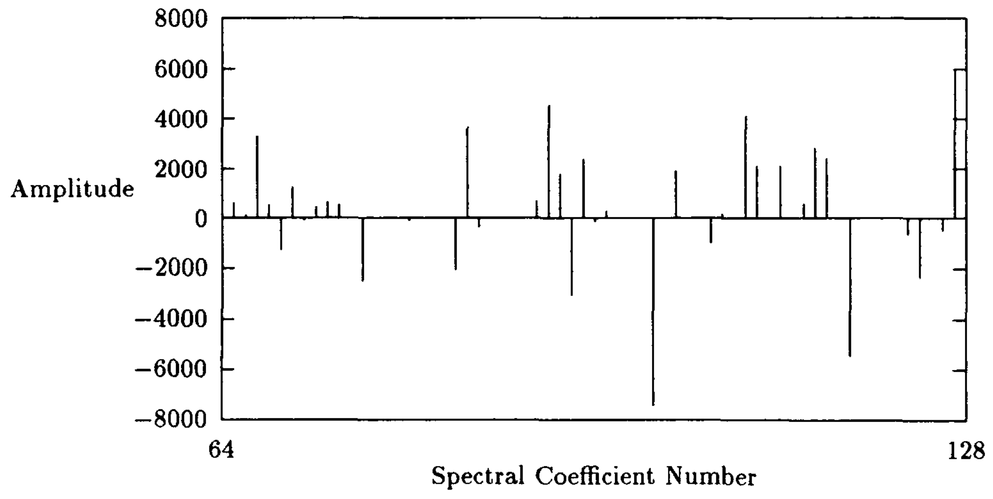

Figure 4.25. Polynomial Fit Spectral Coefficients 64 – 128 of Figure 4.22

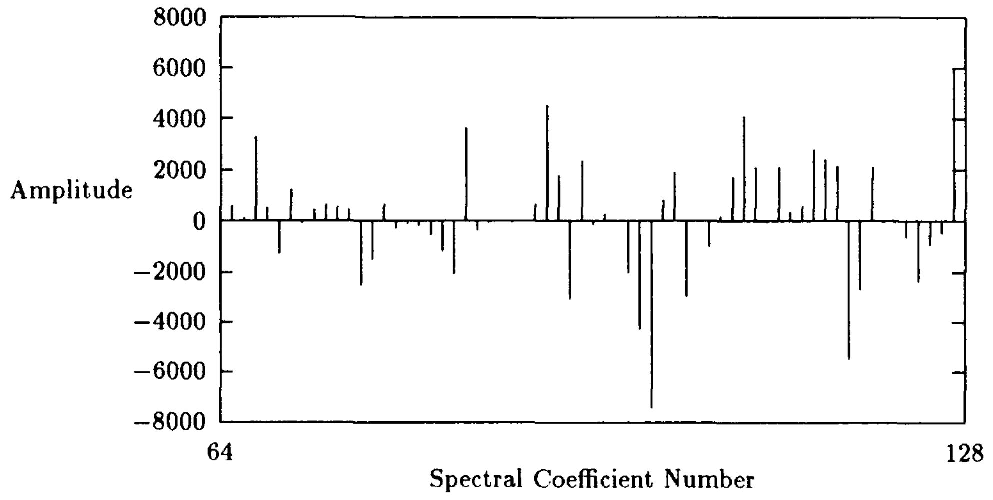

Figure 4.26. Reconstructed Polynomial Fit Spectral Coefficients 64 – 128 of Figure 4.25

4.6.4 Cepstral Processing Homomorphic filtering, or homomorphic deconvolution, is a process designed to separate convolved components of a signal (25:807). Cepstral processing is a specific homomorphic filtering technique with numerous applications, including speech (6:1428). The term "cepstrum" comes from the interchange of each letter in the first syllable of "spectrum". Using this same naming convention, terms such as quefrency and liftering correspond to frequency and filtering in the cepstral domain. The power cepstrum is generally defined as the power spectrum of the logarithm of the power spectrum of the function, and is used in detection or estimation of signal parameters (6:1429). Another technique, the complex cepstrum, retains the phase information of the data, and can therefore be used in reconstruction applications (6:1430). The equation for the complex cepstrum (25:782) is given in Equation 4.3 below:

$$c_m = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log \mid S(e^{j\omega}) \mid e^{j\omega} d\omega. \tag{4.3}$$

The disadvantage of this technique is the complexity involved in calculating the complex logarithm. The transform used in this type of system is a Fourier transform. The technique discussed in this section involves application of cepstral-like techniques using the GLOT.

The coefficients of the GLOT are real valued, yet in magnitude and phase representation, they contain a phase term of either zero or 180 deg which must be maintained to reconstruct the signal. Recognizing that the phase term needed to be maintained, but that it could only be one of two specific values, the following definition was derived to govern a cepstral-like transformation as it relates to the GLOT :

$$\mathcal{L}_{j,k}(x) = \mathcal{G}_F[\ln \mid (\mathcal{G}_F(s(x_{j,k})) \mid]. \tag{4.4}$$

where $s(x_{j,k})$ is the data, $\mathcal{G}_F$ is the forward GLOT, and $\mathcal{L}_{j,k}(x)$ are the resulting cepstral-like coefficients. This new transform actually encompasses parts of both the power cepstrum and complex cepstrum described earlier. As in the power cepstrum, a complex logarithm is not required. The magnitude operation in equation 4.4 facilitates this simplified operation. Because the phase information can only be one of two values, it can easily be extracted as part of the algorithm and used in the reconstruction. The phase information is preserved for reconstruction, as in the complex cepstrum given in Equation 4.3, without using the complex logarithm. By tailoring cepstral-type operations specifically to the GLOT, the best of both techniques can be exploited. Given cepstral-like coefficients defined by $\mathcal{L}_{j,k}(x)$, the signal can be reconstructed by :

$$S_{j,k} = \mathcal{G}_I[P_{j,k}\mathcal{G}_I(\mathcal{L}_{j,k}(x))]. \tag{4.5}$$

where $\mathcal{G}_I$ is the inverse GLOT, $P_{j,k}$ is a phase vector correcting for the magnitude scaling in equation 4.4, and $S_{j,k}$ is the reconstructed signal such that $S_{j,k} \approx s(x_{j,k})$.

Using this new decomposition and reconstruction of the signal, various compression techniques of the cepstral coefficients (mocpression techniques ?) can be tested. If this new transform can "pack" more signal information into fewer coefficients, then greater compression can result by retaining only the information packed cepstral coefficients.

This transformation is different from the original GLOT. Therefore, the first test was to use all cepstral coefficients in reconstruction of the signal. The cepstral coefficients in one window are given in Figure 4.27. The output signal was an exact copy of the input signal under these conditions. In this test, the perfect reconstruction criteria was satisfied, and correct implementation of the theory outlined above was also validated. The next two sections detail two methods employed in an effort to determine a high quality, low bit rate technique for speech processing.

4.6.4.1 Cepstral Liftering The cepstral coefficients near the origin (low quefrency) define the spectral envelope (26:203-204). Based on this observation, one technique which can be employed is to retain the first x coefficients to be used in the reconstruction of the signal, where x is can vary from one to the total number of cepstral coefficients. The compression ratio is then given by x/N, where N is the number of samples (and cepstral coefficients) in each interval. This technique involves liftering (filtering) the low quefrency cepstral coefficients. Figure 4.27 displays low quefrency coefficients having higher amplitude than the rest of the cepstral coefficients, and liftering will pass these important coefficients. The same four compression percentages used earlier were tested here. The out-

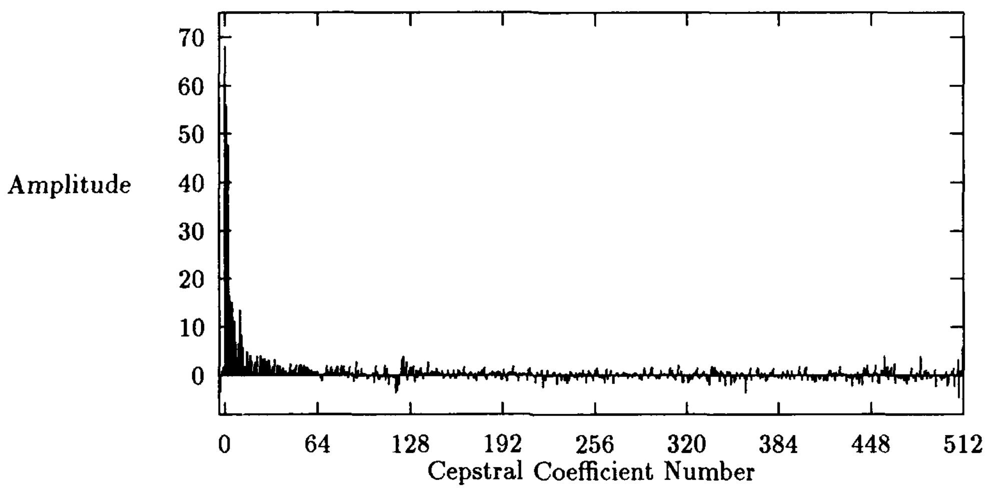

Figure 4.27. Non-liftered Non-thresholded Cepstral Coefficients in One Window on Interval [4096, 4608]

puts for 85.0% and 98.1% compression of the cepstral coefficients are given in Figures 4.28 and 4.29, respectively.

In all previous plots, higher quality recorded outputs had waveforms that matched the input waveform to a high degree. Conversely, output waveforms which did not match the input waveform were of poor audio quality. This trend does not indicate, however, that a correlation always exists between reconstructed speech quality and waveshape. Comparing Figures 4.28 and 4.29 indicates strong similarity between the outputs. When listening to these two outputs, however, the output in Figure 4.28 was of significantly higher audio quality.

The cepstral lifter technique distorted the speakers voice slightly at the 85.0% compression level. There were no extraneous tones in the output as in the case of the absolute threshold output. The cepstral liftering seemed to garble the speaker's voice, and it was not as clear as the absolute threshold output at this compression level. At higher compression ratios, the cepstral lifter technique provided much higher quality outputs when compared to the absolute threshold method. The transition from 85.0% to 92.5% did not

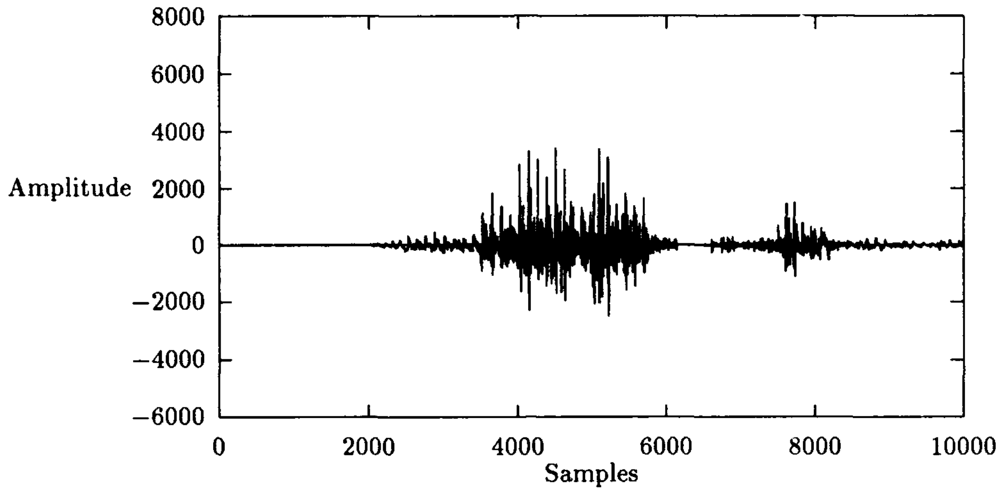

Figure 4.28. Output Waveform After Cepstral Liftering and Performing 85.0% Compression

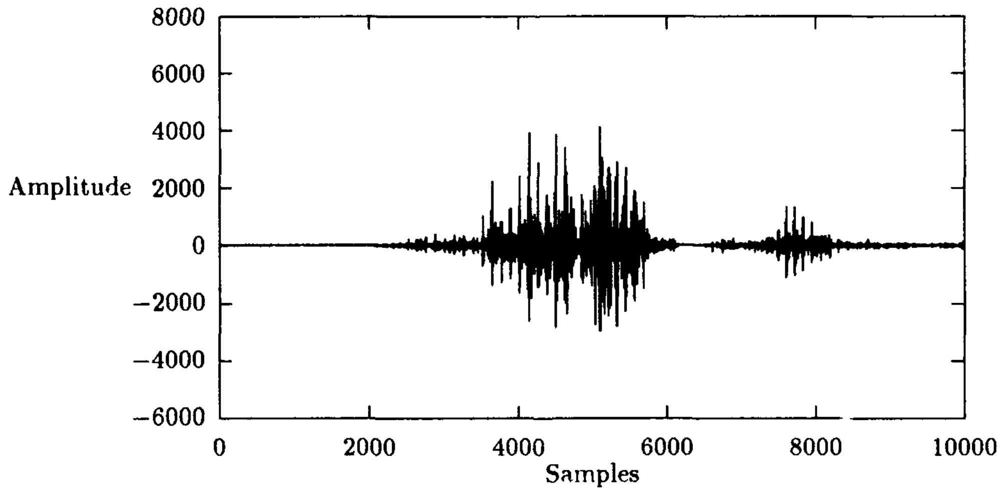

Figure 4.29. Output Waveform After Cepstral Liftering and Performing 98.1% Compression

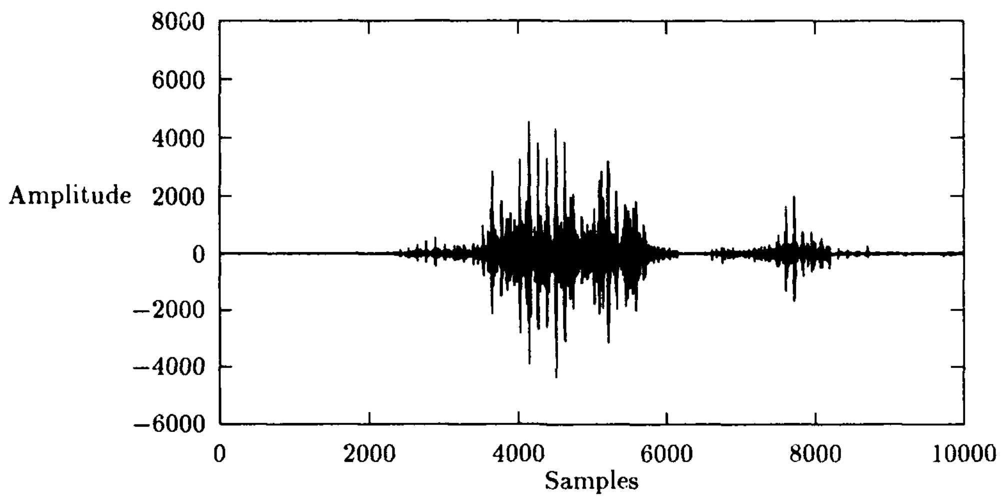

Figure 4.30. Output Waveform After Cepstral Liftering and Performing 99.1% Compression

audibly detract from the quality of the reconstructed output, and further compression to 96.2% and 98.1% were the highest quality reproductions at these levels compared to any of the previous techniques. This success prompted another transformation for a bit rate of 300 bit/sec as computed in Section 4.6.1. 300 bits/sec corresponds to using less than one percent of the coefficients in reconstruction of the signal! Figure 4.30 is a plot of the output resulting from this reconstruction. The speech was recognizable at this level, but contained significant noise. The waveform is very similar to the waveforms in Figures 4.28 and 4.29, although the audio quality of all three is different.

This technique was different from all other techniques in that the number of coefficients retained in each window remained constant. Less overhead would be required to pass the necessary reconstruction information through a communications channel, since the location and number of coefficients in each window is preordained. In all other schemes, the number of coefficients and their unique position in frequency would need to be passed in addition to the value of the coefficients. The disadvantage of this technique is that the same number of coefficients are used to represent windows containing no speech as those

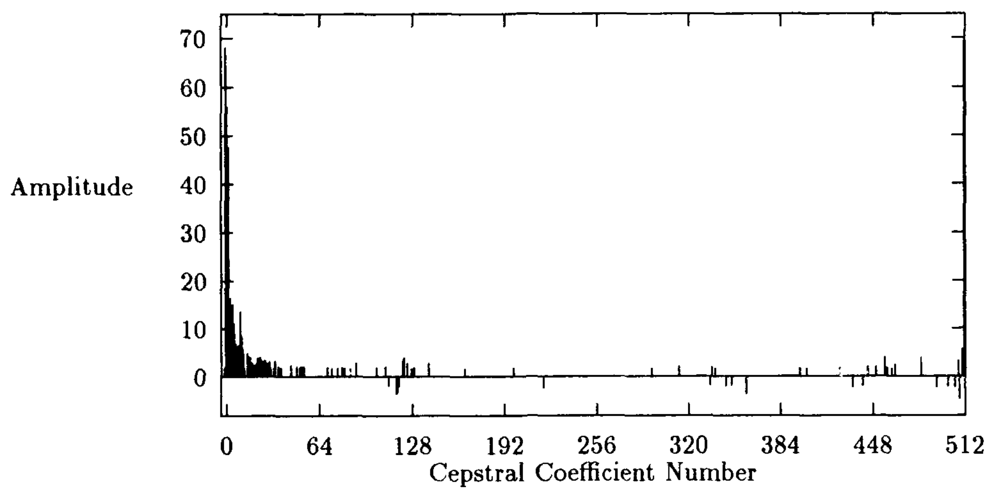

Figure 4.31. Thresholded Cepstral Coefficients (85.0%) in One Window on Interval [4096, 4608]

containing speech in the entire interval. If 12 coefficients are liftered from each window, and there are 10 windows where no speech is present, then 120 cepstral coefficients are retained which have no information. To overcome this indiscriminate coefficient retention, the technique of the next section was investigated.

4.6.4.2 Cepstral Thresholding Rather than liftering the low quefrency cepstral coefficients from each window, a threshold was used to retain a percentage of the highest amplitude coefficients. Because the low quefrency coefficients are generally of higher amplitude (see Figure 4.27), this technique has the effect of low pass liftering. The difference here is that more low quefrency cepstral coefficients will be retained in speech intervals and less will be retained in non speech intervals. This technique is completely analogous to the spectral thresholding technique of Section 4.6.1. The coefficients retained in one example window for compression percentages 85.0% and 98.1% are given in Figures 4.31 and 4.32, respectively. Notice that some higher quefrency coefficients are retained in Figure 4.31, due to the higher cepstral threshold. When the threshold is increased to achieve 98.1% compression, however, the result is a basically a low pass lifter

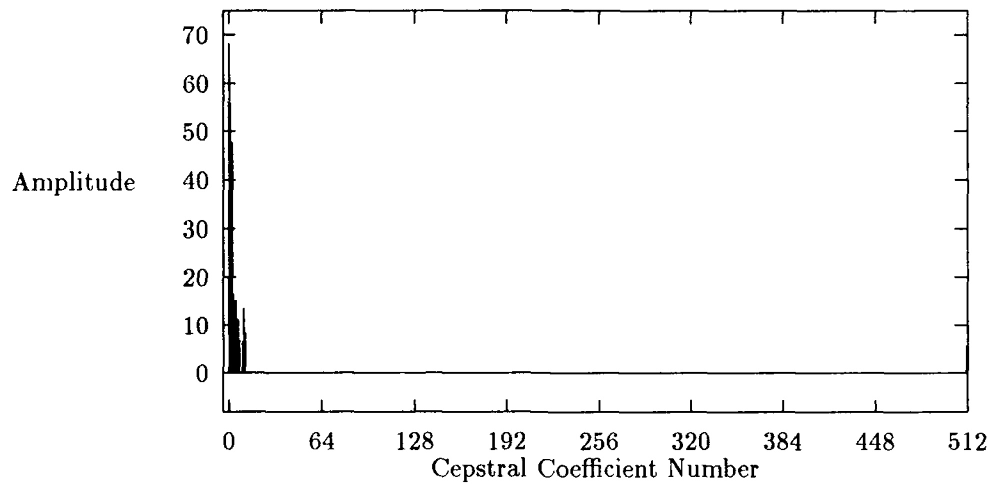

Figure 4.32. Thresholded Cepstral Coefficients (98.1%) in One Window on Interval [4096, 4608]

of the first 12 cepstral coefficients. Figure 4.33 gives only the low quefrency coefficients of Figure 4.33. The only difference between a 12 coefficient low pass lifter and the technique resulting in Figure 4.33 is that one of the 12 low quefrency coefficients did not meet the threshold, and was zeroed out along with the higher quefrency coefficients. The output resulting from the 85.0% compression level, given in Figures 4.34, was closer in form to the input than any of the liftering outputs. At higher compression ratios, however, the output waveforms were very similar to the liftering output waveforms given above.

This technique detracted from the speaker's voice quality less than the liftering technique. At the 85.0% compression level, however, the absolute threshold technique still had the best quality reproduction. At higher compression ratios (above 98%), cepstral thresholding outperformed all other techniques tested.

Because of the success of this technique at high compression ratios, the 99.1% compression level was tested as in Section 4.6.4.1 above. Figures 4.35 and 4.36 give the thresholded cepstral coefficients of one window and resulting output waveform for this compression level. Only the first 64 cepstral coefficients are shown in Figure 4.35, since all others

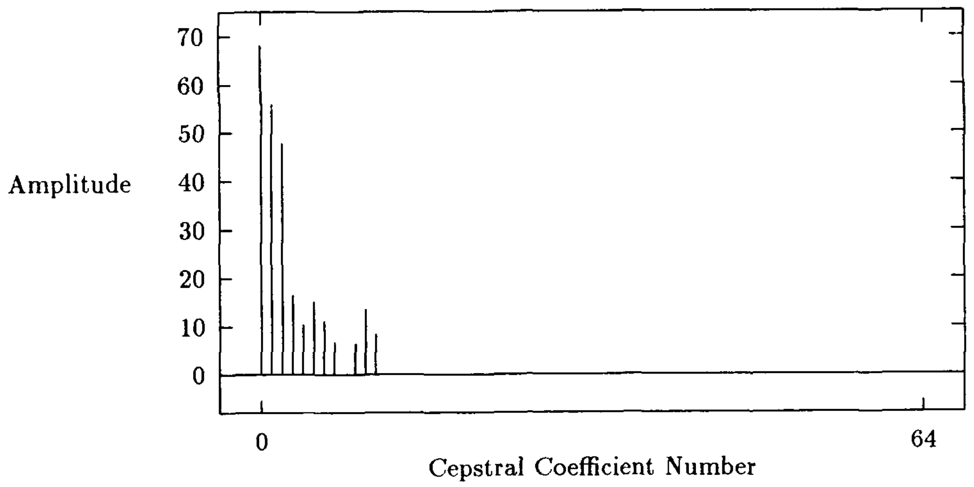

Figure 4.33. First 64 Thresholded Cepstral Coefficients (98.1%) in One Window on Interval [4096, 4608]

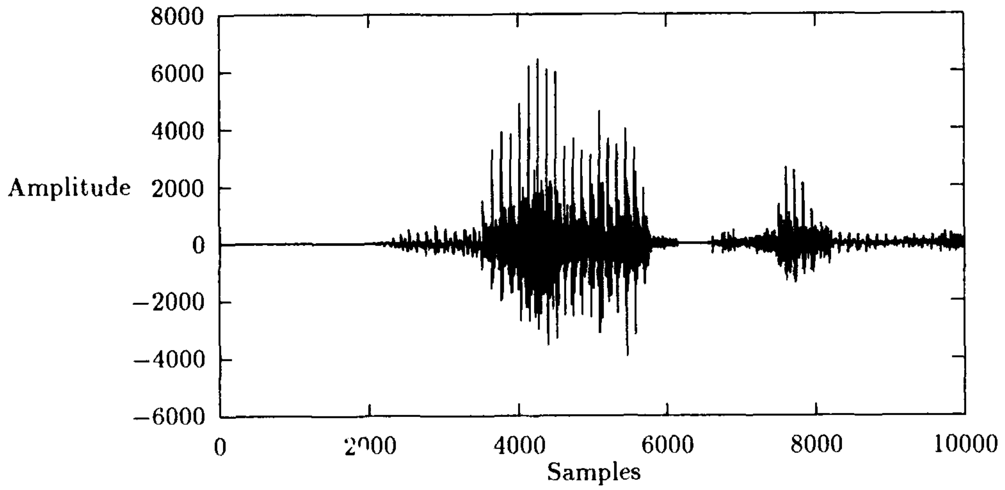

Figure 4.34. Cepstral Thresholding Output Waveform with 99.1% Compression

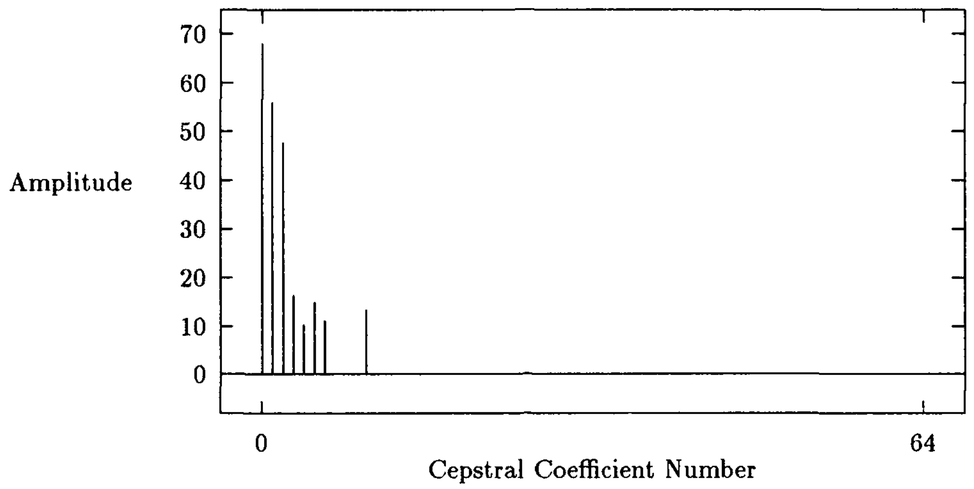

Figure 4.35. First 64 Thresholded Cepstral Coefficients (99.1%) in One Window on Interval [4096, 4608]

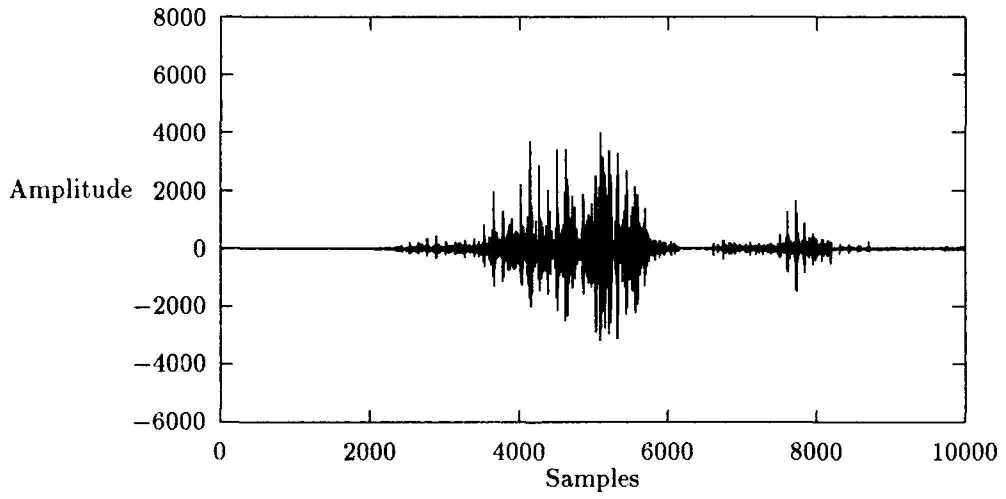

Figure 4.36. Cepstral Thresholding Output Waveform with 99.1% Compression

are zero. The audio quality at this high compression level was better than the quality of the analogous liftering output. The speech was understandable, although noise was present which altered the speaker's voice.

4.6.5 Summary All of the techniques given in Section 4.6 were analyzed to gain a better understanding of the GLOT, and define potential areas which merit further research toward development of a low bit rate speech encoder. Due to the amount of previous research in the area of speech compression, and the complexities of speech processing, the body of experiments performed in this thesis is not sufficient to define such an encoder. The following chapter will focus on potential follow-on topics toward this goal, as well as summarize the results of this thesis.

V. Conclusion

5.1 Summary and Conclusions

This thesis was the incipiency of research regarding the generalized lapped transform and its application to the processing of speech signals. As such, the initial research emphasis was proper validation and implementation of the theory. Many areas hold promise for future research, and some of these topics are outlined in the next section. First, however, a summary of the results of this thesis will be provided which have laid the foundation for subsequent efforts.

The first stage in validating the theory behind the generalized lapped transform was the development of a prototype system using the digital signal processing tool, PC-DSP. Use of a prototype system, especially when implementing an untested algorithm, can significantly reduce the overall time to develop a working system. The use of PC-DSP was especially helpful in making minor corrections to theory, and quickly validating certain theoretical predictions. Specifically, the use of PC-DSP experimentally demonstrated that perfect reconstruction of a signal was possible when using all spectral coefficients in the reconstruction algorithm. Further, testing of the eventual software program was facilitated, because expected results from each stage of the algorithm were known a priori.

The most significant coding task and result was the development and testing of a fast Fourier transform (FFT) algorithm. Although developed specifically to support the generalized lapped transform , this Ada based algorithm is of use in any digital signal processing application. Most FFT routines are limited to radix 2 application; however, this general purpose algorithm can process data of arbitrary length. The timing analysis in Chapter 3 also reveals that this Ada based routine, and the Ada language in general, is viable for digital signal processing applications.

The software program outlined in Appendix A provides a decomposition/reconstruction algorithm based on a sinusoidal basis function and windows with variable length and percent overlap with adjacent windows. Initial testing confirmed that perfect reconstruction of a signal was achievable when using all spectral coefficients in the reconstruction, thereby satisfying the first research goal outlined in Chapter 1. This result assumes the signal is defined outside the interval of interest. Due to the success of the finite length signal processing technique outlined in Sections 2.2 and 4.3.1.6, perfect reconstruction was obtained for the first and last data segments as well, even though the signal was undefined on one adjacent interval. This technique is the first nonexpansive lapped transform based solution which achieves perfect reconstruction of a finite length signal, and has potential Air Force applications in areas such as image coding, where the signal of interest is at the border of the image. The second research goal involving finite length signals was satisfied based on this solution.

Previous research in the area of lapped transforms has demonstrated perfect reconstruction for data defined outside the interval of interest using a $50/k\%$ overlap, and mathematically derived conditions for a more general size overlap (19) (8). This thesis implemented, for the first time, a lapped transform algorithm based on the work of Suter and Oxley (33) which achieved perfect reconstruction of a speech signal with a one point overlap between adjacent windows. This result is significant because a transform based on a one point overlap requires half the processing of a transform based on a 50% overlap.

The independent interval processing program (See Appendix C) provides a method of achieving perfect reconstruction of a signal without overlapping adjacent intervals. This result is an extension of the finite length signal solution, and previous lapped transform algorithms have not addressed this method of processing a signal. This result developed late in the thesis cycle. Consequently, only minimal testing and experimentation for this specific implementation of the generalized lapped transform was performed. Despite limited experimentation, this processing method has merits which warrant further research (See Section 5.2.1).

The software program of Appendix A was thoroughly tested to ensure operation in accordance with the theory outlined in Chapter 2. After validating the theory, a set of experiments were performed to gain insight into the applicability of the transform to speech processing, and specifically speech compression. These experiments were based on various spectral coefficient processing techniques, including absolute thresholding, relative thresholding, polynomial fitting, and cepstral processing. The absolute thresholding technique involved setting a threshold to determine which spectral coefficients would be used in reconstruction of the signal. Any coefficient of amplitude greater than the threshold was retained, while any coefficient not meeting the threshold was set to zero. This technique provided the best results in terms of audio quality at compression ratios up to 85%, although some perceptible noise was apparent in the reconstructed output. The relative threshold technique used the same retention criteria, although the threshold was unique to each window. This technique had the advantage of better reproduction of the beginning or ending of words, but overall, was inferior to the absolute threshold technique in terms of noise. The polynomial fitting technique involved representation of the spectral coefficients with a polynomial curve defined by the relative maxima and minima coefficients. This technique provided outputs equal in audio quality to the absolute threshold technique for 85% compression. In addition, the polynomial fitting technique provided better speech reproduction at word boundaries (analogous to the relative threshold technique). This technique was inherently limited to a compression ratio of 85%. The most promising technique investigated was the cepstral processing technique, which produced an intelligible output when using less than one percent of the cepstral coefficients in reconstruction of the signal. Two variations on cepstral processing were investigated. The first method employed a low pass lifter (filter) to the cepstral coefficients, retaining the first x coefficients. Varying x defined a compression percentage. The second method was analogous to the absolute threshold method, only in the cepstral domain. A percentage of cepstral coefficients were retained based on a threshold level. The second cepstral technique produced a better quality audio output compared to the first method. The trade-off is that the first method requires overhead to send the position of retained cepstral coefficients in each window. The second method involves reconstruction based on a set number and position of low quefrency cepstral coefficients, and only the coefficient amplitude need be encoded. All of the methods described were investigated to determine the applicability of the GLOT to speech processing as outlined in Goal three of Chapter 1. Further research is needed (see Section 5.2.3) to determine if these techniques can be exploited to define a complete low bit rate speech encoder.

5.2 Follow-On Research Areas

5.2.1 Independent Interval Processing The solution to the finite length signal problem, extended to provide independent processing of adjacent intervals, has many advantages which render it attractive for further study. Previous lapped transform and filter bank approaches require an overlap with adjacent data. One advantage of this new approach is that processing interval j is not dependent on processing interval $j + 1$. If real-time processing is required for a given communication system, this advantage could become important. Also, error control is facilitated with this new approach, since errors at the boundary are not propogated into the adjacent interval. Further, total decoupling of adjacent intervals allows for parallel processing of data. This could also become important in real-time system applications. Finally, processing M data points with a periodically extended signal requires only M storage locations throughout the transformation. Processing M data points with an overlapped signal requires $M + l$ locations, where l is the sum of both overlaps. Both methods result in M coefficients in the transformation; however, the previous overlapping method requires a larger buffer. Further investigation of this technique is required to determine which signal processing and communications applications may benefit from its use.

5.2.2 Performance of the GLOT in the Presence of Noise The effects of noise were not considered in this thesis. In any practical communication system, however, noise can have a significant impact on performance. Follow-on efforts involving any application must include an analysis of the effects of noise. The theory predicts that any noise could be perfectly reconstructed, along with the output signal, when using all spectral coefficients. Perfect reconstruction of the noise may not be desirable, however, and methods of filtering and coding the spectral coefficients which eliminate noise must be investigated. If the noise is somehow convolved with the signal of interest, it may be liftered (filtered) in the cepstral domain, since cepstral analysis is based on separation of convolved signals (6). In general, the performance of the generalized lapped transform in noise will depend on the specific type of noise present and the specific application.

5.2.3 Cepstral Processing The experiments discussed in Section 4.6.4 indicate the potential for a low bit rate speech coder based on the generalized lapped transform and cepstral-like processing. Advances in fast Fourier transform algorithms, such as the one described in Chapter 3, has made sophisticated cepstral techniques practical for real-time applications such as speech (6:1441). Additional research is required to determine the best technique for compressing the ceptral coefficients, including the number of bits required per coefficient and the amount of overhead resulting from position or phase information required for reconstruction.

5.2.4 Variable Orthonormal Bases The theory provided by Suter and Oxley (33) allows for different orthonormal representations in adjacent windows, *even though these windows overlap.* This concept has not been presented in previous research. The program implemented as part of this research did not directly address multiple orthonormal representations, as the sinusoidal basis defined in equation 2.3 was exclusively used. Expanding the implementation program to encompass variable orthonormal bases, such as an orthonormal polynomial, must be investigated. A signal which is not sinusoidal in some interval may require many coefficients for representation. Potentially, the number of coefficients can be decreased if another basis set more closely matched to the signal is chosen in that interval. For example, a speech signal is generally sinusoidal in nature during speech intervals, but slowly varying in non-speech intervals. Combining the features of variable size windows and variable orthonormal bases could result in an effective speech compression algorithm. A sinusoidal basis would be used to represent the speech signal in voiced regions, while another basis set could be used to represent the signal in unvoiced regions. Although the current implementation does not recognize these different parameter intervals, Switzer (34) has developed a program to determine the voiced/unvoiced speech boundaries. Combining the generalized lapped transform (generalized to include additional orthonormal bases) and the work of Switzer is an area which merits further research.

Appendix A. *Ada Source Code*

This appendix provides all source code used during the development and testing of the generalized lapped transform.

A.1 Main Program

```
--------------------------------------------------------------------------
--------------------------------------------------------------------------
with Text_Io;
with Complex_Pkg;
use Complex_Pkg;
with Math_Lib;
use Math_Lib;
with Vector_Package;
use Vector_Package;
with Type_Package;
use Type_Package;
with Print_Package;
use Print_Package;
with FFT_Pack;
use FFT_Pack;


  procedure GLOT is

    package Integer_Io is new Text_Io.Integer_Io(integer);
    package Float_Io is new Text_Io.Float_Io(float);

    Max_Data_Length                 : constant := 100_000;
    Max_Partitions                  : constant := 500;

    Big_Daddy_Data_Vector           : Real_Vector (1..Max_Data_Length);
    Big_Daddy_Partition_Vector      : Partition_Vector (0..Max_Partitions);

    Actual_Data_Length              : integer := 0;
    Actual_Partitions               : integer := 0;

    Deriv_Name,
    Alpha_Name,
    Outfile_Name,
    Difference_Name                 : string (1..30);

    Deriv_Name_Length,
    Alpha_Name_Length,
    Outfile_Name_Length,
    Difference_Name_Length          : integer range 1..30;

    Info_File                       : Text_Io.File_Type;
```

```
--Get_File_Data prompts the user for the prepared info file name
--and then reads in the data in the following order
--    (1) Input Data Filename
--    (2) Derivative Output Filename
--    (3) Alpha Output Filename
--    (4) Output Data Filename
--    (5) Input Output Difference Filename
--    (6) General Info Filename
--    (7) Number of Points in the Data Vector
--    (8) Number of required partitions
--    (9) - (?) loop for input given in (8) and get
--    Boundary point then Overlap for each partition

    procedure Get_File_Data     (  Derivative_Filename   : in out string;
                                   Derivative_Length     : in out integer;
                                   Alpha_Filename        : in out string;
                                   Alpha_Length          : in out integer;
                                   Output_Filename       : in out string;
                                   Outname_Length        : in out integer;
                                   Diff_Filename         : in out string;
                                   Diff_Length           : in out integer;
                                   Infofile              : in out Text_Io.File_Type;
                                   Partitions            : in out Partition_Vector;
                                   Number_Of_Partitions  : in out integer;
                                   Data                  : in out Real_Vector;
                                   Points                : in out integer ) is

        -- Filenames, lengths must be in out so they can
        -- be written to info file.
        -- Number_Of_Partitions and Points must be in out
        -- Parameters because they are used in a loop to
        -- Read in the appropriate amount of data

        Infile,
        Datafile                       : Text_Io.File_Type;
        Temp_Input                     : integer;
        Data_Filename                  : string(1..30);
        Data_Filename_Length           : integer;
        In_Filename                    : string(1..30);
        In_Filename_Length             : integer;
        Info_Filename                  : string(1..30);
        Info_Length                    : integer;

        Counter                        : integer := 0;

    begin

        Text_Io.New_Line;
        Text_Io.put("What is the name of the file containing ");
        Text_Io.put("the prepared information ? = ");
        Text_Io.Get_Line ( In_Filename, In_Filename_Length);
        Text_Io.Open(Infile, Text_Io.In_File, In_Filename(1..In_Filename_Length));
```

```
- - Now I call in the required user data
Text_Io.Get_Line (Infile, Data_Filename, Data_Filename_Length);
Text_Io.Get_Line (Infile, Derivative_Filename, Derivative_Length);
Text_Io.Get_Line (Infile, Alpha_Filename, Alpha_Length);
Text_Io.Get_Line (Infile, Output_Filename, Outname_Length);
Text_Io.Get_Line (Infile, Diff_Filename, Diff_Length);
Text_Io.Get_Line (Infile, Info_Filename, Info_Length);
Integer_Io.get (Infile,Points);
Integer_Io.get (Infile,Number_Of_Partitions);
------------------------------------------------------------------
Text_Io.Create(Infofile, Text_Io.Out_File, Info_Filename(1..Info_Length));

Text_Io.put (Infofile,"This information describes the outputs");
Text_Io.put (Infofile," produced from prepared data file = ");
Text_Io.Put_Line (Infofile,In_Filename(1..In_Filename_Length));
Text_Io.New_Line (Infofile);

Text_Io.put (Infofile,"Input Data Filename > ");
Text_Io.Put_Line (Infofile,Data_Filename(1..Data_Filename_Length));
Text_Io.New_Line (Infofile);

Text_Io.put(Infofile,"Derivative Filename = ");

Text_Io.Put_Line(Infofile,Derivative_Filename (1..Derivative_Length));
Text_Io.New_Line(Infofile);

Text_Io.put(Infofile,"Alphas Filename = ");
Text_Io.Put_Line(Infofile,Alpha_Filename(1..Alpha_Length));
Text_Io.New_Line(Infofile);

Text_Io.put(Infofile,"Output Data Filename = ");
Text_Io.Put_Line(Infofile,Output_Filename(1..Outname_Length));
Text_Io.New_Line(Infofile);

Text_Io.put(Infofile,"In Out Difference Filename = ");
Text_Io.Put_Line(Infofile,Diff_Filename(1..Diff_Length));
Text_Io.New_Line(Infofile);

Text_Io.put(Infofile,"Number of Data Points = ");
Integer_Io.put(Infofile,Points,1);
Text_Io.New_Line(Infofile); Text_Io.New_Line(Infofile);

Text_Io.put(Infofile,"Number of Partitions = ");
Integer_Io.put(Infofile,(Number_Of_Partitions + 1),1);
Text_Io.New_Line(Infofile);

------------------------------------------------------------------
for jin 1..Number_Of_Partitions loop
    Integer_Io.get(Infile,Partitions(j).Boundary);
    Integer_Io.get(Infile,Partitions(j).Overlap);
end loop;
Text_Io.Close(Infile);

- - now I call in the numbers for the vector Data
Text_Io.Open(Datafile,Text_Io.In_File, Data_Filename(1..Data_Filename_Length));
for j in 1..Points loop
```

```
        Integer_Io.Get(Datafile,Temp_Input);
        Data(j) := float(Temp_Input); - - for integer inputs
    end loop;
    Text_Io.Close(Datafile);

    Partitions(0).Boundary := 1;
    Partitions(0).Overlap := 0;

    Partitions(Number_Of_Partitions + 1).Boundary := Points + 1;
    Partitions(Number_Of_Partitions + 2).Boundary := Points + Partitions(1).Boundary;
    Partitions(Number_Of_Partitions + 1).Overlap := 0;
    Partitions(Number_Of_Partitions + 2).Overlap := 0;

    Number_Of_Partitions := Number_Of_Partitions + 2;

    - - write the first partition to the end of the data file
    - - for wrap around processing
    Counter := 1;
    for j in (Points + 1)..(Points + Partitions(1).Boundary) loop
        Data(j) := Data(Counter);
        Counter := Counter + 1;
    end loop;

    Points := Points + Partitions(1).Boundary;

end Get_File_Data;

------------------------------------------------------------------
------------------------------------------------------------------
- Multiply_By_Window
------------------------------------------------------------------
------------------------------------------------------------------
procedure Multiply_By_Window ( Full_Data_Segment : in Real_Vector;
                               Folded_In_Data    : in out Real_Vector;
                               Window_j          : in out Real_Vector) is

    Pi                      : constant := 3.141592654;
    EPSILON_j               : integer  := Folded_In_Data'first - Full_Data_Segment'first;
    EPSILON_j_plus_1        : integer  := Full_Data_Segment'last - Folded_In_Data'last;
    A_j                     : integer  := Folded_In_Data'first;
    A_j_plus_1              : integer  := Folded_In_Data'last;
    Window_Boundary_Value   : constant float := 1.0/sqrt(2.0);

begin

    - - This is the basic window - also need complex window
-----------------------CREATE WINDOW---------------------------
    - - Window defined in Section 2.1.2.1, equation 2.8.

    - - left rising edge
    if EPSILON_j = 0 then
        Window_j(A_j) := Window_Boundary_Value;
    else
        for x in (A_j - EPSILON_j)..(A_j + EPSILON_j) loop
            Window_j(x) := sin((Pi/(4.0*float(EPSILON_j))) * (float(x-(A_j-EPSILON_j))));
        end loop;
        end if;
        - - center
        for x in (A_j + EPSILON_j + 1)..(A_j_plus_1 - EPSILON_j_plus_1 - 1)
        loop
            Window_j(x) := 1.0;
        end loop;

        - - right falling edge
        if EPSILON_j_plus_1 = 0 then
            Window_j(A_j_plus_1) := Window_Boundary_Value;
        else
            for x in (A_j_plus_1 - EPSILON_j_plus_1).. (A_j_plus_1 + EPSILON_j_plus_1) loop
                Window_j(x) := cos((Pi/(4.0*float(EPSILON_j_plus_1))) *
                                   (float(x-(A_j_plus_1 -EPSILON_j_plus_1))));
            end loop;

        end if;

----------------------Multiply By Window----------------------
        for x in (A_j)..(A_j + EPSILON_j) loop
            Folded_In_Data(x) := Full_Data_Segment(x) * Window_j(x) -
                                    Full_Data_Segment(2*A_j - x) * Window_j(2*A_j - x);
        end loop;

        for x in (A_j + EPSILON_j + 1)..(A_j_plus_1 - EPSILON_j_plus_1 - 1) loop
            Folded_In_Data(x) := Full_Data_Segment(x) * Window_j(x);
        end loop;

        for x in (A_j_plus_1 - EPSILON_j_plus_1)..(A_j_plus_1) loop
            Folded_In_Data(x) := Full_Data_Segment(x) * Window_j(x) +
                            Full_Data_Segment(2*A_j_plus_1 - x) * Window_j(2*A_j_plus_1 - x);
        end loop;




    end Multiply_By_Window;

--------------------------------------------------------------------
--------------------------------------------------------------------
- - Compute_Coefficients
- -
- - This procedure takes in data from 0 - N or A_j - A_j+1 (N+1 points) and
- - returns coefficients stored from 0 - N-1 or A_j - (A_j+1 - 1) (N points)
- - Nth point is used in calculations but is not a valid out point
- - It will be used to store the zero'th coefficient in the next data segment
--------------------------------------------------------------------
--------------------------------------------------------------------
    procedure Compute_Coefficients ( Data_Segment : in out Real_Vector;
                                     Deriv_Output  : in out Text_Io.File_Type) is

        Two_N          : integer := 2 * (Data_Segment'length - 1);
        End_FFT_Data   : integer := Data_Segment'first + Two_N - 1;
        FFT_Data       : Complex_Vector(Data_Segment'first..End_FFT_Data);
    begin
       - - (1) Even Extend Data_Segment
       - - (See Vector_Package-Complex_Package and Section 2.1.2.4, Step 5).
       FFT_Data := Complex_Of(Even_Extend(Data_Segment));

       - - (2) Perform Inverse FFT (See FFT_Pack and Section 2.1.2.4, Step 6).
       FFT (FFT_Data, True); - - includes 1/N scaling

       for x in Data_Segment'range loop
          Float_io.put(Deriv_Output,FFT_Data(x).real);
          Text_Io.New_Line(Deriv_Output);
       end loop;

       - - (3.a) Assign zero coefficient to first point in Data_Segment.
       Data_Segment(Data_Segment'first) := FFT_Data(FFT_Data'first).Real * sqrt(float(Two_N));

       - - (3.b) Assign coefficients 1 to N-1 back to Data_Segment after integration
       - - (See Section 2.1.2.4, Step 7).
       for k in (Data_Segment'first + 1)..(Data_Segment'last-1) loop
          Data_Segment(k) := Data_Segment(k-1) + (2.0 * sqrt(float(Two_N)) * FFT_Data(k).Real);
       end loop;

    end Compute_Coefficients;
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
- - Reconstruction FFT
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------

  procedure Reconstruction_FFT ( Input_Data : in Real_Vector;
                                 Output_Data : out Real_Vector ) is

       Pi               : constant := 3.141592654;
       X                : float    := 0.0; - - float counter
       Scale_Factor,
       Complex_Factor   : Complex;

       Two_N            : integer  := 2 * Input_Data'length;
       Two_N_Float      : float    := float(Two_N);
       End_FFT_Data     : integer  := Input_Data'first + Two_N - 1;
       FFT_Data         : Complex_Vector(Input_Data'first..End_FFT_Data);

    begin

       - - (1) Odd Extend Input_Data
       - - (See Vector_Package, Complex_Package and Section 2.1.3, Step 1).
       FFT_Data := Complex_Of(Odd_Extend(Input_Data));

       - - (2) Perform Inverse FFT (See FFT_Pack and Section 2.1.3, Step 2).
       FFT(FFT_Data, True);

       X := 0.0;
       Scale_Factor.real := sqrt(Two_N_Float);
       Scale_Factor.imag := 0.0;

       for l in Output_Data'range loop
```

```
            Complex_Factor.real := Cos(Pi*X/Two_N_Float);
            Complex_Factor.imag := Sin(Pi*X/Two_N_Float);
            FFT_Data(l) := Scale_Factor * Complex_Factor * FFT_Data(l);
            X := X + 1.0; - - float counter
        end loop;

        - - assign scaled imaginary part to Output_Data 1..N
        for l in Output_Data'range loop
            Output_Data(l) := FFT_Data(l).imag;
        end loop;

    end Reconstruction_FFT;

------------------------------------------------------------------------
------------------------------------------------------------------------
- - Divide_By_Windows (See Section 2.1.3, Step 3).
------------------------------------------------------------------------
------------------------------------------------------------------------

    procedure Divide_By_Windows ( Data_Segment   : in out Real_Vector;
                                  Window_Right   : in Real_Vector;
                                  Boundary_Point: in integer ) is

        Counter                 : integer := 0;
        Window_Left             : Real_Vector(Window_Right'range);
        Temp_Data_Segment       : Real_Vector(Data_Segment'range) := Data_Segment;

    begin

        Window_Left := Reverse_Assignment(Window_Right);

        - - Calculate Data_Segment(A_j - e_j .. A_j - 1)
        Counter := Window_Right'last;
        for i in (Window_Right'first)..(Boundary_Point-1) loop
            Data_Segment(i) := Window_Left(i) * Temp_Data_Segment(i) -
                               Window_Left(Counter) * Temp_Data_Segment(Counter);
            Counter := Counter - 1;
        end loop;

        - - Calculate Data_Segment(A_j)
        Data_Segment(Boundary_Point) := Temp_Data_Segment(Boundary_Point) /
                               (2.0 * Window_Left(Boundary_Point));

        - - Calculate Data_Segment(A_j + 1 .. A_j + e_j - 1)
        Counter := Boundary_Point;
        for i in (Boundary_Point + 1)..(Window_Right'last - 1) loop
            Counter := Counter - 1;
            Data_Segment(i) := Window_Right(Counter) * Temp_Data_Segment(Counter) +
                               Window_Right(i) * Temp_Data_Segment(i);
        end loop;

        - - Calculate Data_Segment(A_j + e_j .. A_j-1 - e_j-1)
        - - for i in Window_Right'last..Temp_Data_Segment'last loop
        - -     Data_Segment(i) := Temp_Data_Segment(i);
                - - Window values here are unity so no action required
                - - If different scaling was required this would have to be
            -- rewritten - best way would be to pass in another slice
            -- from window and giving it a new name as input to Div_By_Win
        -- end loop;

    end Divide_By_Windows;

------------------------------------------------------------------
------------------------------------------------------------------
-- Do_Work
------------------------------------------------------------------
------------------------------------------------------------------

    procedure Do_Work ( Data                 : in Real_Vector;
                        Partitions           : in Partition_Vector;
                        Derivfile_String     : in String;
                        Alphafile_String     : in String;
                        Outfile_String       : in String;
                        Diff_File_String     : in String;
                        Infofile             : in out Text_Io.File_Type) is

        Pi                              : constant := 3.141592654;

        Transformed_Data , T_Data       : Real_Vector(Data'range);
        Output_Data                     : Real_Vector(Data'range);
        Window                          : Real_Vector(Data'range);
        Median_Data, Median2_Data       : Real_Vector(Data'range);

        Start_Data, End_Data,                   -- A_j and A_j+1
        Start_Window, End_Window,               -- A_j - e_j and A_j+1 + e_j+1
        Last_Output_Point,                      -- A_j+1 - e_j+1
        Last_Window_Point,                      -- A_j + e_j
        Counter,
        Alpha_Compression_Counter       : integer := 0;
        Two_N,
        Temp_Nth_Value                  : float := 0.0;
        -- Used because Nth value needed in two intervals
        -- and otherwise would be overwritten
        Derivfile,Alphafile,
        Outfile, Diff_File              : Text_Io.File_Type;
        Alpha_Threshold                 : float := 0.0;
        Percent_Remaining               : float := 0.0;

    begin
        Text_Io.Create (Derivfile, Text_Io.Out_File, Derivfile_String);
        Text_Io.Create (Alphafile,Text_Io.Out_File, Alphafile_String);
        Text_Io.Create (Outfile,   Text_Io.Out_File, Outfile_String );
        Text_Io.Create (Diff_File, Text_Io.Out_File, Diff_file_String);

        Text_Io.New_Line;
        -- This threshold is optional depending on the application
        Text_Io.put("What is the desired coefficient threshold ? ");
        Float_Io.get(Alpha_Threshold);
        Text_Io.New_Line;

        for j in 0..(Partitions'last-1) loop

            begin -- each segment is transformed within this loop
```

```
-----------------------Assign j Dependent Values-----------------------

          Start_Data := Partitions(j).Boundary;
          End_Data := Partitions(j+1).Boundary;
          Start_Window := Partitions(j).Boundary - Partitions(j).Overlap;
          End_Window := Partitions(j+1).Boundary + Partitions(j+1).Overlap;

          Two_N := float(2 * (End_Data-Start_Data));

          Last_Output_Point := End_Data - Partitions(j+1).Overlap - 1;
          Last_Window_Point := Start_Data + Partitions(j).Overlap;
          Temp_Nth_Value := Transformed_Data(End_Data);

-----------------------------------------------------------------------

          Text_Io.put("Working.... Processing Data Segment");
          Integer_Io.put(j+1);
          Text_Io.New_Line;

          Multiply_By_Window ( Data(Start_Window..End_Window),
                               Transformed_Data(Start_Data..End_Data),
                               Window(Start_Window..End_Window));

             -- Steps 2 and 3 of Coefficient Evaluation (Section 2.1.2.4).

             -- Window added as passed parameter only so
             -- it does not have to be recalculated
             -- in Divide_By_Windows

          Counter := 0;
          for l in Start_Data..End_Data loop
             Transformed_Data(l) := Transformed_Data(l) * sin((Pi * float(Counter))/(Two_N));
             Counter := Counter + 1; -- Counter goes from 0 to N
          end loop;

             -- Step 4 of Coefficient Evaluation (Section 2.1.2.4).

          Compute_Coefficients ( Transformed_Data(Start_Data..End_Data),
                                 Derivfile);
                -- This routine performs
                -- (1) an even extension of the data,
                -- (2) an inverse FFT,
                -- (3) assigns coefficients 0 - N-1 to Transformed_Data
                -- Steps 5, 6, and 7 of Coefficient Evaluation (Section 2.1.2.4).

-----------------------Print Alphas for each segment to file-----------------------

          for x in Start_Data..(End_Data-1) loop
             float_io.put(Alphafile,Transformed_Data(x));
             text_io.new_line(Alphafile);
          end loop;
-----------------------------------------------------------------------

          Transformed_Data(End_Data) := Temp_Nth_Value;
          -- Transformed_Data will not be changed again in
```

```
        -- this loop. This assignment replaces the Nth value
        -- back to Transformed_Data so it can be used
        -- as the Start_Data point in the following segment

        Reconstruction_FFT ( Transformed_Data
                              (Start_Data..(End_Data-1)),
                              Output_Data((Start_Data+1)..End_Data));
        -- This routine performs
        -- (1) an odd extension of the data,
        -- (2) an inverse FFT,
        -- (3) assigns the scaled imaginary result to
        -- the Output_Data values 1 - N or A_j + 1 to A_j+1

        -- This indicates that data values 0 - N
        -- create alphas from 0 - N-1 which in turn are used
        -- to reconstruct data values from 1 - N
        -- WOW !!!

        -- Steps 1 and 2 of Reconstruction (Section 2.1.3).

        Divide_By_Windows ( Output_Data(Start_Window..Last_Output_Point),
                             Window(Start_Window..Last_Window_Point),
                             Partitions(j).Boundary);

        if j= (Partitions'last - 1) then
          begin
            Counter := 1;
            for k in Start_Data..Last_Window_Point loop
              Output_Data(Counter) := Output_Data(k);
              Counter := Counter + 1;
            end loop;
          end; -- block
        end if;

    end; -- block
end loop; -- j loop

Text_Io.Close(Alphafile);

Text_io.Put_Line("Printing output to file...");


for x in Partitions(0).Boundary..(Partitions(Partitions'last - 1).Boundary - 1) loop
    Integer_Io.put(Outfile,(Integer(Output_Data(x))));
    Text_Io.New_Line(Outfile);
    Integer_Io.put(Diff_File,(Integer(Output_Data(x) - Data(x))));
    Text_Io.New_Line(Diff_File);
end loop;

Text_io.New_Line(Infofile);
Text_io.put(Infofile,"The threshold was ");

Float_io.put(Infofile,Alpha_Threshold,2,2,0);
Text_io.New_Line(Infofile);Text_io.New_Line(Infofile);
Text_io.put(Infofile,"No. of zeroed coefficients was ");
Integer_io.put(Infofile,Alpha_Compression_Counter,2);
```

```
    Text_io.New_Line(Infofile);Text_io.New_Line(Infofile);
    Text_io.put(Infofile,"The Percent_Remaining := 100.0 * (1.0 -
           (float(Alpha_Compression_Counter) /
           float((Partitions(Partitions'last-1).Boundary - 1))));
    Float_io.put(Infofile, Percent_Remaining,2,5,0);
    Text_io.New_Line(Infofile);

    Text_Io.Close(Outfile);
    Text_Io.Close(Diff_File);

    Text_Io.New_Line;
    Float_Io.put(Alpha_Threshold,3,5,0);
    Text_Io.New_Line;
    Integer_Io.put(Alpha_Compression_Counter,3); Text_Io.New_Line;
    Float_Io.put(Percent_Remaining,3,5,0);
    Text_Io.New_Line;

end Do_Work;
---------------------------------------------------------------------
---------------------------------------------------------------------

begin - - main

    Get_File_Data ( Deriv_Name, Deriv_Name_Length,
                    Alpha_Name, Alpha_Name_Length,
                    Outfile_Name, Outfile_Name_Length,
                    Difference_Name, Difference_Name_Length,
                    Info_File,
                    Big_Daddy_Partition_Vector, Actual_Partitions,
                    Big_Daddy_Data_Vector, Actual_Data_Length);

         - - By sending the correct size arrays into Do_Work, the array
         - - attributes can be used to determine the size rather than
         - - passing another parameter
    Do_Work ( Big_Daddy_Data_Vector(1..Actual_Data_Length),
              Big_Daddy_Partition_Vector (0..Actual_Partitions),
              Deriv_Name(1..Deriv_Name_Length),
              Alpha_Name(1..Alpha_Name_Length),
              Outfile_Name(1..Outfile_Name_Length),
              Difference_Name(1..Difference_Name_Length),
              Info_file);
end; - - main
---------------------------------------------------------------------
```

A.2 Package Type_Package

```
package Type_Package is

    type Complex is
                record
                        Real : float;
                        Imag : float;
                end record;

    type Partition_Record is
                record
                        Boundary : integer;
                        Overlap : integer;
                end record;

    type Partition_Vector is array (integer range <>) of Partition_Record;

                - - Partition_Vector is an array of Records
                - - Each record specifies the boundaries
                - - and the number of points of overlap

    type Direction is (Increasing, Decreasing, Undefined);
    type Phase_Type is (Pos, Neg, Zero);
    type Phase_Vector is array (integer range <>) of Phase_Type;
    - - used only for polynomial fit compression technique outlined in Section 4.6.3

    type Real_Vector is array (integer range <>) of float;
    type Complex_Vector is array (integer range <>) of Complex;

                - - By using an unconstrained arrays <> I will be able to
                - - pass array slices in and out of procedures which are sub-
                - - procedures to Do_Work. The only arrays global to the
                - - entire program are Big_Daddy_Data_Vector and
                - - Big_Daddy_Partition_Vector
                - - All other arrays are slices tailored to user
                - - input in Get_Parameters.

end Type_Package;
```

A.3 Package FFT_Pack

The code for the FFT Package specification and body are contained in Appendix A.

A.4 Package Vector_Package

A.4.1 Vector_Package Specification

```
with Text_Io;
with Type_package;
use  Type_Package;
```

```
with Complex_Pkg;
use  Complex_Pkg;

package vector_package is

-----------------------------------------------------------------------------

    function Odd_Extend(In_Vector : in Real_Vector) return Real_Vector;

    function Even_Extend(In_Vector : in Real_Vector) return Real_Vector;

    function Reverse_Assignment ( In_Vector : in Real_Vector) return Real_Vector;

end vector_package;
```

A.4.2 Vector_Package Body

```
-----------------------------------------------------------------------------
package body vector_package is

package integer_io is new text_io.integer_io(integer);

    function Even_Extend(In_Vector : in Real_Vector) return Real_Vector is

          - - See Section 2.1.2.4, Step 5.

          Two_N               : integer := 2 * (In_Vector'length -1);
          End_Extended_Data   : integer := In_Vector'first + Two_N - 1;
          Extended_Data       : Real_Vector(In_Vector'first..End_Extended_Data);
          Counter             : integer := In_Vector'last - 1;

        begin
          Extended_Data(Extended_Data'first..In_Vector'last) := In_Vector;
          for i in (In_Vector'last + 1)..Extended_Data'last loop
             Extended_Data(i) := In_Vector(Counter);
             Counter          := Counter - 1;
          end loop;

          return Extended_Data;

        end Even_Extend;

-----------------------------------------------------------------------------

    function Odd_Extend(In_Vector : in Real_Vector) return Real_Vector is

          - - See Section 2.1.3, Step 1.

          Two_N               : integer := 2 * In_Vector'length;
          End_Extended_Data   : integer := In_Vector'first + Two_N - 1;
          Extended_Data       : Real_Vector(In_Vector'first..End_Extended_Data);
          Counter             : integer := In_Vector'last;

        begin
          Extended_Data(Extended_Data'first..In_Vector'last) := In_Vector;
        for i in (In_Vector'last + 1)..Extended_Data'last loop
            Extended_Data(i) := - In_Vector(Counter);
            Counter := Counter - 1;
        end loop;

        return Extended_Data;

    end Odd_Extend;
```

```
    function Reverse_Assignment(In_Vector : in Real_Vector) return Real_Vector is

        Counter             : integer := In_Vector'first;
        Reversed_Vector     : Real_Vector(In_Vector'range);

    begin
        for y in reverse In_Vector'range loop
            Reversed_Vector(Counter) := In_Vector(y);
            Counter := Counter + 1;
        end loop;

        return Reversed_Vector;
    end Reverse_Assignment;
```

```
end vector_package;
```

A.5 Package Complex_Pkg

A.5.1 Complex_Pkg Specification

```
with Type_Package;
use Type_Package;

package Complex_Pkg is

    - - arithmetic operations

    function "+" ( A,B : Complex) return Complex;
    function "-" ( A,B : Complex) return Complex;
    function "*" ( A,B : Complex) return Complex;
    function "*" ( R   : float;
                   C   : Complex) return Complex;
    function "/" ( A   : Complex;
                   B   : float)      return Complex;
    function "/" ( A,B : Complex) return Complex;
    function Negative (A : Complex) return Complex;

    - - conversion operations

    function Complex_Of (R,I : float) return Complex;
```

```
     function Complex_Of (R_Vector : Real_Vector) return Complex_Vector;
     function Complex_Of (R : float) return Complex;
     function Conjugate (A : Complex) return Complex;
     procedure Set_Equal ( A : in Complex; B : out Complex);

     - - I/O operations

     procedure Get (A : out Complex);
     procedure Put (A : in Complex);

end Complex_Pkg;
```

A.5.2 Complex_Pkg Body

```
with Math_Lib;
use Math_Lib;
with Text_Io;
use Text_Io;

package body Complex_Pkg is

     package Flt_Io is new Float_Io(float);
     use Flt_Io;

     function "+" (A,B : Complex) return Complex is
              Result : Complex;
         begin
              Result.Real := A.Real + B.Real;
              Result.Imag := A.Imag + B.Imag;
              Return Result;
         end "+";

     function "-" (A,B : Complex) return Complex is
              Result : Complex;
         begin
              Result.Real := A.Real - B.Real;
              Result.Imag := A.Imag - B.Imag;
              Return Result;
         end "-";

     function "*" (A,B : Complex) return Complex is
              Result : Complex;
         begin
              Result.Real := (A.Real*B.Real) - (A.Imag*B.Imag);
              Result.Imag := (A.Real*B.Imag) + (A.Imag*B.Real);
              Return Result;
         end "*";

     function "*" (R : float; C : Complex) return Complex is
              Result : Complex;
         begin
              Result.Real := R * C.Real;
              Result.Imag := R * C.Imag;
              Return Result;
         end "*";
```

```
function "/" (A : Complex; B : float) return Complex is
        Result : Complex;
    begin
        Result.Real := A.Real / B;
        Result.Imag := A.Imag / B;
        Return Result;
    end "/";

function "/" (A,B : Complex) return Complex is
        Result : Complex;
        Squares : float;
    begin
        Squares := A.Real ** 2 + B.Imag ** 2;
        Result.Real := (A.Real * B.Real + A.Imag * B.Imag) / Squares;
        Result.Imag := (A.Imag * B.Real - A.Real * B.Imag) / Squares;
        Return Result;
    end "/";

function Negative (A : Complex) return Complex is
        Result : Complex;
    begin
        Result.Real := -A.Real;
        Result.Imag := -A.Imag;
        Return Result;
    end Negative;

function Complex_Of (R,I : float) return Complex is
        Result : Complex;
    begin
        Result.Real := R;
        Result.Imag := I;
        Return Result;
    end Complex_Of;

function Complex_Of (R_Vector : Real_Vector) return Complex_Vector is
        Result : Complex_Vector(R_Vector'range);
    begin
        for i in R_Vector'range loop
            Result(i).Real := R_Vector(i);
            Result(i).Imag := 0.0;
        end loop;
        Return Result;
    end Complex_Of;

function Complex_Of (R : float) return Complex is
        Result : Complex;
    begin
        Result.Real := R;
        Result.Imag := 0.0;
        Return Result;
    end Complex_Of;

function Conjugate (A : Complex) return Complex is
        Result : Complex;
    begin
```

```
            Result.Real := A.Real;
            Result.Imag := - A.Imag;
            return Result;
        end Conjugate;

    procedure Set_Equal ( A : in Complex; B : out Complex) is
        begin
            B.Imag := A.Imag;
            B.Real := A.Imag;
        end Set_Equal;

    procedure Get (A : out Complex) is
        begin
            Get(A.Real);
            Get(A.Imag);
        end Get;
    procedure Put (A : in Complex) is
        begin
            Put ("(");
            Put (A.Real, 1,2,0);
            Put (",");
            Put (A.Imag,5,2,0);
            Put(")");
        end Put;

end Complex_Pkg;
```

Appendix B. *Polynomial Fit Compression Technique Code*

The following procedures were added to support the polynomial fitting compression technique described in Section 4.6.3.

B.1 Procedure Maxima_Minima

```
procedure Maxima_Minima ( Data                          : in out Real_Vector;
                          Threshold                     : in float;
                          First_Pos_Maxima,
                          First_Neg_Maxima,
                          Last_Pos_Maxima,
                          Last_Neg_Maxima               : in out integer;
                          Phase                         : out Phase_Vector) is

    First_Pos_Loop, First_Neg_Loop                      : boolean := true;
    Pos_Slope, Neg_Slope                                : Direction := Undefined;
    Last_Pos_Point_Saved,
    Last_Neg_Point_Saved,
    Last_Pos_Point_Considered,
    Last_Neg_Point_Considered                           : integer := 0;
    First_Trip_Through_Pos_Decreasing_Slope,
    First_Trip_Through_Neg_Increasing_Slope             : boolean := true;
  begin
    First_Pos_Maxima := 0;
    First_Neg_Maxima := 0;
    - - These values are set to zero to be used in this procedure
    - - to keep track of the status of First* and Last*
    Last_Pos_Maxima := 0;
    Last_Neg_Maxima := 0;
    - - These values are set to zero for each segment so they can be checked
    - - upon exit from Maxima_Minima and used as a decision basis for
    - - reconstructing using polynomials

    for j in Data'range loop

      begin - - loop
        if Data(j) > Threshold then

          begin - - positive values

            Phase(j) := Pos;

            if Pos_Slope = Increasing then
              if Data(j) > Data(Last_Pos_Point_Considered) then
                    Data(Last_Pos_Point_Considered) := 0.0;
                    Last_Pos_Point_Considered := j;
              else
                    Last_Pos_Point_Saved := Last_Pos_Point_Considered;
```

```
                Last_Pos_Point_Considered := j;
                Pos_Slope := Decreasing;
                if First_Pos_Maxima /= 0 then
                  Last_Pos_Maxima := Last_Pos_Point_Saved;
                  - - if this point in the algorithm is entered at least
                  - - once Last_Pos_Maxima will change from zero to the
                  - - appropriate number and polynomial reconstruction
                  - - will take place on the positive coefficients.
                  - - This ocurrs when the slope changes from increasing
                  - - to decreasing.
                end if;
            end if;

        elsif Pos_Slope = Decreasing then
            if First_Pos_Maxima = 0 then
              First_Pos_Maxima := Last_Pos_Point_Saved;
            end if;
            if Data(j) < Data(Last_Pos_Point_Considered) then
              Data(Last_Pos_Point_Considered) := 0.0;
              Last_Pos_Point_Considered := j;
            else
              Last_Pos_Point_Saved := Last_Pos_Point_Considered;
              Last_Pos_Point_Considered := j;
              Pos_Slope := Increasing;
            end if;

        elsif Pos_Slope = Undefined then - - 1st or 2cd pos loop
            if First_Pos_Loop then - - 1st pos loop
              Last_Pos_Point_Saved := j;
              First_Pos_Loop := false;
            else - - 2cd pos loop
              if Data(j) > Data(Last_Pos_Point_Saved) then
                Pos_Slope := Increasing;
              else
                Pos_Slope := Decreasing;
              end if;
              Last_Pos_Point_Considered := j;
            end if;
        end if; - - Pos Slope
    end; - - block for positive values

  elsif Data(j) < -Threshold then

    begin - - negative values

        Phase(j) := Neg;

        if Neg_Slope = Increasing then
            if First_Neg_Maxima = 0 then
              First_Neg_Maxima := Last_Neg_Point_Saved;
            end if;

            if Data(j) > Data(Last_Neg_Point_Considered) then
              Data(Last_Neg_Point_Considered) := 0.0;
              Last_Neg_Point_Considered := j;
            else
```

```
                Last_Neg_Point_Saved := Last_Neg_Point_Considered;
                Last_Neg_Point_Considered := j;
                Neg_Slope := Decreasing;
              end if;

            elsif Neg_Slope = Decreasing then
              if Data(j) < Data(Last_Neg_Point_Considered) then
                Data(Last_Neg_Point_Considered) := 0.0;
                Last_Neg_Point_Considered := j;
              else
                Last_Neg_Point_Saved := Last_Neg_Point_Considered;
                Last_Neg_Point_Considered := j;
                Neg_Slope := Increasing;
                if First_Neg_Maxima /= 0 then
                  Last_Neg_Maxima := Last_Neg_Point_Saved;
                  -- See comment above in positive values algorithm.
                end if;
              end if;

            elsif Neg_Slope = Undefined then -- 1st or 2cd neg loop
              if First_Neg_Loop then -- 1st neg loop
                Last_Neg_Point_Saved := j;
                First_Neg_Loop := false;
              else -- 2cd neg loop
                if Data(j) > Data(Last_Neg_Point_Saved) then
                  Neg_Slope := Increasing;
                else
                  Neg_Slope := Decreasing;
                end if;
                Last_Neg_Point_Considered := j;
              end if;
            end if; -- Neg Slope
          end; -- block for negative values

        else
          Data(j) := 0.0;
          Phase(j) := Zero;
        end if;

      end; -- block

    end loop;
  end Maxima_Minima;
```

B.2 Procedure Fit_Slice

```
-- Fit_Slice
------------------------------------------------------------------------

procedure Fit_Slice (Data_Segment : in out Real_Vector;
                     Phase_Segment : in Phase_Vector;
                     fx1_Position  : in integer;
                     Correct_Phase : in Phase_Type;
                     Thresh        : in float ) is

    fx0       : float := Data_Segment(Data_Segment'first);
    fx1       : float := Data_Segment(fx1_Position);
    fx2       : float := Data_Segment(Data_Segment'last);

    Alpha0    : float := fx0;
    -- Either fx0 or Alpha0 could be eliminated
    -- but both are provided for clarity
    Alpha1,
    Alpha2    : float; -- To be calculated;

    x1        : float := float(fx1_position - Data_Segment'first); -- x0 = 0
    x2        : float := float(Data_Segment'last - Data_Segment'first);
    x         : float := 1.0; -- floating loop variable
    Factor    : float := 1.0;

  begin

    Alpha2 := ( fx2 - fx0 - (x2/x1*(fx1-fx0)) ) / (x2*(x2-x1)) ;
    Alpha1 := ( fx1 - fx0 - (Alpha2*x1*x1) ) / x1 ;

    for j in (Data_Segment'first+1)..(Data_Segment'last-1) loop
       if Phase_Segment(j) = Correct_Phase then
          Data_Segment(j) := Alpha0 + (alpha1*x) + (Alpha2*x*x) ;
          -- Check to ensure polynomial fit is not out of allowable range
          if Correct_Phase = Pos then
             if Data_Segment(j) < Thresh then
                Data_Segment(j) := Thresh;
             end if;
          else -- Correct_Phase = Neg then
             if Data_Segment(j) > -Thresh then
                Data_Segment(j) := -Thresh;
             end if;
          end if;

       end if;
       x := x + 1.0; -- x will always = j - Data_Segment'first
    end loop;

  end Fit_Slice;
```

B.3 Procedure Polynomial_Fit

```
--------------------------------------------------------------------------------
-- Polynomial_Fit
--------------------------------------------------------------------------------
  procedure Polynomial_Fit (Data                    : in out Real_Vector;
                            First_Pos_Maxima,
                            First_Neg_Maxima,
                            Last_Pos_Maxima,
                            Last_Neg_Maxima         : in integer;
                            Phase                   : in Phase_Vector;
                            Threshold               : in out float        ) is

      -- Threshold is passed in only to be used in Fit_Slice

      Start                                 : integer := First_Pos_Maxima;
      Middle, Endslice                      : integer := 0;
      The_Last_Point_Saved_Was_A_Min        : boolean := false;

    begin

      if Last_Pos_Maxima /= 0 then
        begin -- positive reconstruction is valid

          for x in (First_Pos_Maxima+1)..Last_Pos_Maxima loop
            if Data(x) > 0.0 then
              -- then the point is either a min or a max

              if The_Last_Point_Saved_Was_A_Min then
                Endslice := x;
                Fit_Slice(Data(Start..Endslice),
                          Phase(Start..Endslice),
                          Middle, Pos, Threshold );
                The_Last_Point_Saved_Was_A_Min := false;
                Start                          := Endslice;

              else                    -- else the last point
                Middle := x;          -- saved was a max
                The_Last_Point_Saved_Was_A_Min := true;

              end if;

            end if;
          end loop; -- positive reconstruction loop

        end; -- pos recon valid block
      end if;

      -- Now handle negative polynomial reconstruction
      The_Last_Point_Saved_Was_A_Min := false; -- re-initialize
      Start := First_Neg_Maxima;

      if Last_Neg_Maxima /= 0 then
        begin -- negative reconstruction is valid

          for x in (First_Neg_Maxima+1)..Last_Neg_Maxima loop
```

```
                if Data(x) < 0.0 then
                   -- then the point is either a min or a max

                   if The_Last_Point_Saved_Was_A_Min then
                      Endslice:= x;
                      Fit_Slice(Data(Start..Endslice),
                                Phase(Start..Endslice),
                                Middle, Neg , Threshold );
                      The_Last_Point_Saved_Was_A_Min := false;
                      Start                          := Endslice;

                   else                   -- else the last point
                      Middle := x;        -- saved was a max
                      The_Last_Point_Saved_Was_A_Min := true;

                   end if;

                end if;
             end loop; -- negative reconstruction loop

          end; -- neg recon valid block
       end if;

   end Polynomial_Fit;
```

Appendix C. *Independent Interval Processing Code Modifications*

The following procedures replace those given in Appendix A for implementation of the independent interval processing outlined in Sections 2.3 and 4.3.6.

C.1 Get_File_Data

```
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- Get_File_Data prompts the user for the prepared info file name
-- and then reads in the data in the following order
--    (1) Input Data Filename
--    (2) Derivative Output Filename
--    (3) Alpha Output Filename
--    (4) Output Data Filename
--    (5) Input Output Difference Filename
--    (6) General Info Filename
--    (7) Number of Points in the Data Vector
--    (8) Number of required partitions
--    (9) - (?) loop for input given in (8) and get
--    Boundary point then Overlap for each partition
-----------------------------------------------------------------------
      -- All code in Procedure Get_File_Data remains the same except the code
      -- following the Datafile input loop which assignes the data to vector Data

         Partitions(0).Boundary := 1;
         Partitions(0).Overlap := 0;

         Partitions(Number_Of_Partitions + 1).Boundary := Points + 1;
         Partitions(Number_Of_Partitions + 1).Overlap := 0;

         Number_Of_Partitions := Number_Of_Partitions + 1;
         Points := Points + Partitions(Number_Of_Partitions).Overlap + 1;
      end Get_File_Data;
```

C.2 *Divide_By_Windows*

```
------------------------------------------------------------------------
------------------------------------------------------------------------
-- Divide_By_Windows (See Section 2.1.3, Step 3).
------------------------------------------------------------------------
------------------------------------------------------------------------

      procedure Divide_By_Windows ( Data_Segment : in out Real_Vector;
                                    Window       : in Real_Vector;
                                    AJ           : in integer ) is

            Counter             : integer := 0;
            Temp_Data_Segment   : Real_Vector(Data_Segment'range) := Data_Segment;
            EJ                  : integer := AJ - Data_Segment'first;
            AJplus1             : integer := Data_Segment'last - EJ;

         begin

            -- (1) boundary
            Data_Segment(AJ) := Temp_Data_Segment(AJ) / (2.0 * Window(AJ));

            -- (2) left overlap
            Counter := AJ;
            for i in (AJ + 1)..(AJ + EJ - 1) loop
               Counter := Counter - 1;
               Data_Segment(i) := Window(Counter) * Temp_Data_Segment(Counter)
                                       + Window(i) * Temp_Data_Segment(i);
            end loop;

            -- (3) middle
            -- for i in ((AJ + EJ)..(AJplus1 - EJ)) loop
               -- Data_Segment(i) := Temp_Data_Segment(i);
               -- Window values here are unity so no action required
               -- If different scaling was required this would have to be
               -- accounted for
            -- end loop;

            -- (4) right overlap
            Counter := Data_Segment'last;
            for i in ((AJplus1 - EJ + 1)..(AJplus1 - 1)) loop
               Counter := Counter -1;
               Data_Segment(i) := Window(i) * Temp_Data_Segment(i) -
                                       Window(Counter)*Temp_Data_Segment(Counter);
            end loop;

         end Divide_By_Windows;
```

C.3 Do_Work

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
- - Do_Work
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

     procedure Do_Work ( Data                : in Real_Vector;
                         Partitions          : in Partition_Vector;
                         Derivfile_String    : in String;
                         Alphafile_String    : in String;
                         Outfile_String      : in String;
                         Diff_File_String    : in String;
                         Infofile            : in out Text_Io.File_Type) is

          Pi                             : constant := 3.141592654;

          Transformed_Data               : Real_Vector(Data'range);
          Window                         : Real_Vector(Data'range);

          aj, aj_plus1, ej,
          aj_less_ej, aj_plus1_less_ej,
          aj_plus_ej, aj_plus1_plus_ej,
          Counter,
          Alpha_Compression_Counter      : integer := 0;
          Two_N,
          Temp_Nth_Value                 : float := 0.0;

          Derivfile,Alphafile,
          Outfile, Diff_File             : Text_Io.File_Type;
          Two_N,
          Alpha_Threshold,
          Percent_Remaining              : float := 0.0;

       begin
          Text_Io.Create (Derivfile, Text_Io.Out_File, Derivfile_String);
          Text_Io.Create (Alphafile,Text_Io.Out_File, Alphafile_String);
          Text_Io.Create (Outfile,  Text_Io.Out_File, Outfile_String );
          Text_Io.Create (Diff_File, Text_Io.Out_File, Diff_file_String);

          Text_Io.New_Line;
          Text_Io.put("What is the desired absolute threshold ? ");
          Float_Io.get(Alpha_Threshold);
          Text_Io.New_Line;

          for j in 0..(Partitions'last-1) loop

             begin - - each segment is transformed within this loop

-----------------------Assign j Dependent Values--------------------------

               ej                       := Partitions(j).Overlap;
```

```
aj                   := Partitions(j).Boundary;
aj_plus_ej           := aj + ej;
aj_less_ej           := aj - ej;

aj_plus1             := Partitions(j+1).Boundary ;
aj_plus1_less_ej     := aj_plus1 - ej;
aj_plus1_plus_ej     := aj_plus1 + ej;

Two_N                := float(2 * (aj_plus1-aj));

---------------------------------------------------------------------------

Text_Io.put("Working.... Processing Data Segment");
Integer_Io.put(j+1);
Text_Io.New_Line;

Transformed_Data(aj_less_ej..aj-1) := Data(aj_plus1_less_ej..aj_plus1-1);
Transformed_Data(aj..aj_plus1-1) := Data(aj..aj_plus1-1);
Transformed_Data(aj_plus1..aj_plus1_plus_ej) := Data(aj..aj_plus_ej);

Multiply_By_Window ( Transformed_Data(aj_less_ej..aj_plus1_plus_ej),
                     Transformed_Data(aj..aj_plus1),
                     Window(aj_less_ej..aj_plus1_plus_ej));

   - - Steps 2 and 3 of Coefficient Evaluation (Section 2.1.2.4).

   - - Window added as passed parameter only so
   - - it does not have to be recalculated
   - - in Divide_By_Windows

Counter := 0;
for l in aj..aj_plus1 loop
   Transformed_Data(l) := Transformed_Data(l)*sin((Pi*float(Counter))/(Two_N));
   Counter := Counter + 1; - - Counter goes from 0 to N
end loop;

   - - Step 4 of Coefficient Evaluation (Section 2.1.2.4).

Compute_Coefficients ( Transformed_Data(aj..aj_plus1),
                       Derivfile);
      - - This routine performs
      - - (1) an even extension of the data,
      - - (2) an inverse FFT,
      - - (3) assigns coefficients 0 - N-1 to Transformed_Data
      - - Steps 5, 6, and 7 of Coefficient Evaluation (Section 2.1.2.4).

-----------------Print Alphas for each segment to file----------------------

for x in aj..(aj_plus1-1) loop
   if abs(Transformed_Data(x)) < Alpha_Threshold then
      Transformed_Data(x) := 0.0;
      Alpha_Compression_Counter := Alpha_Compression_Counter+1;
   end if;
   Float_io.put(Alphafile,Transformed_Data(x));
   Text_io.New_line(Alphafile);
```

```
            end loop;
--------------------------------------------------------------------

            Reconstruction_FFT ( Transformed_Data( aj..(aj_plus1-1)),
                                 Transformed_Data((aj+1)..aj_plus1) );

            Transformed_Data(aj_less_ej..aj) := Transformed_Data(aj_plus1_less_ej..aj_plus1);

            Transformed_Data(aj_plus1+1..aj_plus1_plus_ej) := Transformed_Data(aj+1..aj_plus_ej);

            Divide_By_Windows ( Transformed_Data(aj_less_ej..aj_plus1_plus_ej),
                                Window (aj_less_ej..aj_plus1_plus_ej),
                                Partitions(j).Boundary);

--------------------------------------------------------------------

            for x in (aj..(aj_plus1-1)) loop
               Integer_Io.put(Outfile,(Integer(Transformed_Data(x))));
               Text_Io.New_line(Outfile);
               Integer_Io.put (Diff_File,(Integer(Transformed_Data(x) - Data(x))));
               Text_Io.New_Line(Diff_File);
            end loop;
         end; - - block
      end loop;

      Text_Io.Close(Alphafile);
      Text_Io.Close(Outfile);
      Text_Io.Close(Diff_File);

      Text_io.New_Line(Infofile);
      Text_io.put(Infofile,"The threshold was > ");
      Float_io.put(Infofile,Alpha_Threshold,2,2,0);
      Text_io.New_Line(Infofile);Text_io.New_Line(Infofile);
      Text_io.put(Infofile,"Number of zeroed coefficients was > ");
      Integer_io.put(Infofile,Alpha_Compression_Counter,2);
      Text_io.New_Line(Infofile);Text_io.New_Line(Infofile);
      Text_io.put(Infofile,"The % compression was >");
      Percent_Remaining := 100.0 * (1.0 - (float(Alpha_Compression_Counter) /
            float((Partitions(Partitions'last-1).Boundary - 1))));
      Float_io.put(Infofile, Percent_Remaining,2,5,0);
      Text_io.New_Line(Infofile);

      Text_Io.New_Line;
      Float_Io.put(Alpha_Threshold,3,5,0);
      Text_Io.New_Line;
      Integer_Io.put(Alpha_Compression_Counter,3);
      Text_Io.New_Line;
      Float_Io.put(Percent_Remaining,3,5,0);
      Text_Io.New_Line;

   end Do_Work;
--------------------------------------------------------------------
--------------------------------------------------------------------
```

Vita

Captain Brian Dean Raduenz was born in Minneapolis, Minnesota on 21 March 1966. He graduated from Eveleth High School, Eveleth, MN in 1984, and entered the United States Air Force Academy. He graduated from the Air Force Academy in 1988 with a Bachelor of Science Degree in Electrical Engineering, and a commission as a Second Lieutenant in the U.S. Air Force. Capt Raduenz was then assigned to Hanscom Air Force Base, Bedford, MA, where he served as a project manager on the Modular Control System program (ESD/TC) until April, 1991. He entered the Air Force Institute of Technology in May, 1991. Upon graduation, Capt Raduenz will be assigned to Wright-Patterson Air Force Base, ASC/EN.

Permanent address: 3306 Cedar Island Drive
Eveleth, Minnesota 55734

Bibliography

1. Akansu, Ali N. and Frank E. Wadas. "On Lapped Orthogonal Transforms," *IEEE Transactions on Signal Processing*, *SP-40*(2):439–443 (February 1992).
2. Auscher, Pascal, et al. "Local Sine and Cosine Bases of Coiffman and Meyer." *Wavelets - A Tutorial in Theory and Applications* edited by C. K. Chui, 237–256, Academic Press, 1992.
3. Beylkin, G., et al. "Fast Wavelet Transforms and Numerical Algorithms I," *Communications on Pure and Applied Mathematics*, *XLIV*:141–183 (1991).
4. Beylkin, G., et al. "Wavelets in Numerical Analysis." *Wavelets and Their Applications* edited by M. B. Ruskai, et al., 181–210, Boston, MA: Jones and Bartlett, 1992.
5. Bradley, Jonathan N., et al. "Reflected Boundary Conditions for Multirate Filter Banks." *Proceedings of the IEEE-SP International Symposium on Time-Frequency and Time-Scale Analysis*. 307–310. 4-6 October 1992.
6. Childers, Donald G., et al. "The Cepstrum: A Guide to Processing." *Proceedings of the IEEE*. 1428–1443. October 1977.
7. Coifman, Ronald, "Test of the Bell." Private Communication, April 1992.
8. Coifman, Ronald et Meyer, Yves. "Remarques sur l'analyse de Fourier *à* fenêtre," *C.R. Academie Sci. Paris*, *t.312*(Serie I):259–261 (1991).
9. Daubechies, Ingrid. "Orthonormal Bases of Compactly Supported Wavelets," *Comm. Pure and Applied Math*, *41*:909–966 (1988).
10. de Queiroz, Ricardo L. "Subband Processing of Finite Length Signals." *Proceedings of International Conf. on Accoustics, Speech, and Signal Processing*. IV–613 - IV–616. 23-26 March 1992.
11. Dick, David. "Despite Resistence, Ada Gains Respectability," *Signal Magazine*, *45*(11):92–94 (1991).
12. Ferguson, Warren E. Jr. "A Simple Derivation of Glassman's General N Fast Fourier Transform," *Computers and Mathematics with Applications*, *8*(6):401–411 (1982).
13. Glassman, J. A. "A Generalization of the Fast Fourier Transform," *IEEE Trans. Comput.*, *C-19*:105–116 (1970).
14. Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE, 66*. 51–83. January 1978.
15. Kabrisky, Matthew. Thesis Discussions, August 1992.
16. Karlsson, Gunnar and Martin Vetterli. "Extension of Finite Length Signals for Sub-Band Coding," *Signal Processing*, 7(2):161–168 (June 1989).
17. Mallat, Stêphane and Zhifeng Zhang. *Matching Pursuits with Time-Frequency Dictionaries*. Technical Report 619, Courant Institute of Mathematical Sciences, October 1992.

18. Malvar, Henrique S. "Lapped Transforms for Efficient Transform/Subband Coding," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *38*(6):969–978 (June 1990).

19. Malvar, Henrique S. "Extended Lapped Transforms: Properties, Applications, and Fast Algorithms," *IEEE Transactions on Signal Processing*, *40*(11):2703–2714 (November 1992).

20. Malvar. Henrique S. *Signal Processing with Lapped Transforms*. Boston : Artech House, 1992.

21. Malvar, Henrique S. and David H. Staelin. "The LOT: Transform Coding Without Blocking Effects," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *37*(4):553–559 (April 1989).

22. NIST. *The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus (TIMIT): Training and Test Data and Speech Header Software*, October 1990.

23. Nuri, Veyis and Roberto H. Bamberger. "A Theoretical Framework for the Analysis and Design of Size-Limited Multirate Filter Banks." *Proceedings of the IEEE-SP International Symposium on Time-Frequency and Time-Scale Analysis*. 311–314. 4-6 October 1992.

24. Oktay, Alkin. *PC-DSP*. New Jersey: Prentice Hall, 1990. $3\frac{1}{2}$" IBM Version.

25. Oppenheim, Alan V. and Ronald W. Schafer. *Discrete Time Signal Processing*. New Jersey: Prentice Hall, 1989.

26. Parsons, Thomas W. *Voice and Speech Processing*. McGraw Hill, 1987.

27. Princen, John P. and Alan Bernard Bradley. "Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *ASSP-34*(5):1153–1161 (October 1986).

28. Quackenbush, Schuyler R., et al. *Objective Measures for Speech Processing*. New Jersey: Prentice Hall, 1988.

29. Raduenz, Brian D., et al. "Analysis of an Ada Based Version of Glassman's General *N* Point Fast Fourier Transform," *Computers and Mathematics with Applications* (Submitted July 1992, Accepted October 1992).

30. Singleton, R. C. "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Trans. Audio Electroacoust.*, *AU-17*:93–103 (1969).

31. Stremler, Ferrel G. *Introduction to Communication Systems* (2nd Edition). Addison-Wesley Publishing Co., 1982.

32. Suter, Bruce W. Thesis Discussions, January-August 1992.

33. Suter, Bruce W. and Mark E. Oxley. "On Variable Overlapped Windows and Weighted Orthonormal Bases," *IEEE Transactions on Signal Processing* (Submitted April 1992).

34. Switzer, Shane. *Frequency Domain Speech Coding*. MS thesis, AFIT/GE/ENG/91D, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

35. Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. New Jersey: Prentice Hall, October 1992.

36. Van Loan, Charles. *Computational Frameworks for the Fast Fourier Transform*. Philidelphia: Society for Industrial and Applied Mathematics, 1992.

37. Vetterli, Martin and Didier Le Gall. "Perfect Reconstruction FIR Filter Banks: Some Properties and Factorizations," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *37*(7):1057-1071 (July 1989).

38. Weinstein, Clifford J. "Opportunities for Advanced Speech Processing in Military Computer-Based Systems," *Proceedings of the IEEE*, *79*(11):1627–1639 (November 1991).

December 1992 Masters' Thesis

DIGITAL SIGNAL PROCESSING USING LAPPED TRANSFORMS
WITH VARIABLE PARAMETER WINDOWS
AND ORTHONORMAL BASES

Brian D. Raduenz

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GE/ENG/92D-30

Jon Sjogren, Ph.D.
AFOSR/NM
Bolling AFB, DC 20332-6448

Distribution Unlimited

This thesis develops and evaluates a number of new concepts and tools for the analysis of signals using variable overlapped windows and orthonormal bases. Windowing, often employed as a spectral estimation technique, can result in irreparable distortions in the transformed signal. By placing conditions on the window and incorporating it into the orthonormal representation, any signal distortion resulting from the transformation can be eliminated or cancelled in reconstruction. This concept is critical to the theory underpinning this thesis. As part of this evaluation, a tensor product based general N-point fast Fourier transform algorithm was implemented in the DOD standard language, Ada. The most prevalent criticism of Ada is slow execution time. This code is shown to be comparable in execution time performance to the corresponding FORTRAN code. Also, as part of this thesis, a new paradigm is presented for solving th^ finite length data prob'^m associated with filter banks and lapped transforms. This result could have significant importance in many Air Force applications, such as processing images in which the objects of interest are near the borders. Additionally, a limited number of experiments were performed with the coding of speech. The results indicate the lapped transform evaluated in this thesis has potential as a low bit rate speech coder.

Lapped Transform, GLOT, Fast Fourier Transform 110

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL